Developing Multithreaded Applications: A Platform Consistent Approach

March 2003

Terms of Use

Contents

FIGURES

TABLES

EXAMPLE CODE

1 Overview

Motivation

The objective of this series, which is comprised of this overview and four parts, is to provide guidelines for developing efficient multithreaded applications across Intel® architecture-based symmetric multiprocessors (SMP) and/or systems with Hyper-Threading Technology. An application developer can use the advice contained in this series to improve multithreading performance and to minimize unexpected performance variations on current as well as future SMP architectures built with Intel® processors.

This first version of this documentation provides general advice on multithreaded performance. Hardware-specific optimizations have deliberately been kept to a minimum. In future versions, topics covering hardware-specific optimizations will be added for developers who are willing to sacrifice portability for higher performance.

Prerequisites

Readers should have programming experience in a high-level language, preferably C, C++, and/or Fortran, though many of the recommendations in this document also apply to languages such as Java*, C#, and Perl. Readers must also understand basic concurrent programming and be familiar with one or more threading methods, preferably OpenMP*, POSIX threads (also referred to as Pthreads), or the Win32* threading API.

Scope

The main objective of these documents is to provide a quick reference to design and optimization guidelines for multithreaded applications on Intel® platforms. They are not intended to serve as a textbook on multithreading, nor do they represent a porting guide to Intel platforms.

Organization and Author Attribution

The “Developing Platform Consistent Threaded Applications” series covers topics ranging from general advice applicable to any multithreading method, to usage guidelines for Intel® software products, as well as API-specific issues. While designed as part of a series, each included chapter contains a discrete discussion of an important threading issue and can be read separately.

The included chapters and author attribution for each, are as follows:

Chapter	Scope	Contributing Authors
Chapter 1, **Overview**	This series overview	Bill Magro
Chapter 2, **Intel® Software Development Products**	This chapter describes how to use Intel software products to develop, debug, and optimize multithreaded applications	Bruce Greer, Clay Breshears, Judi Goldstein, Martyn Corden, Phil Kerly, and Vasanth Tovinkere
Chapter 3, **Application Threading**	This chapter covers general topics in parallel performance and occasionally refers to API-specific issues	Aaron Coday, Bill Magro, Clay Breshears, Henry Gabb, Prasad Kakulavarapu, Sanjiv Shah, and Vasanth Tovinkere
Chapter 4, **Synchronization**	The sections in this chapter discuss techniques to mitigate the negative impact of synchronization on performance	Grant Haab, Henry Gabb, Prasad Kakulavarapu and Vasanth Tovinkere
Chapter 5, **Memory Management**	Threads add another dimension to memory management that should not be ignored. This chapter covers memory issues that are unique to multithreaded applications	Clay Breshears, Jay Hoeflinger, Paul Petersen, and Phil Kerly

The user is free to download the entire series as a whole, or to download or read each chapter of interest as the need arises. Cross-references to related topics are provided throughout.

Series Conventions

'This series' refers to the above five chapters. Topics within chapters are referred to as 'sections'. Cross-references are in outline notation, combining chapter and section numbers.

2 Intel® Software Development Products

Intel software development products enable developers to rapidly thread their applications, assist in debugging, and tune multithreaded performance on Intel processors. The product suite supports multiple threading methods, listed here in increasing order of complexity – automatic parallelization, compiler-directed threading with OpenMP*, and manual threading using standard libraries such as Pthreads and the Win32* threading API.

This section introduces the components of the Intel® software development suite by presenting a high-level overview of each product and its key features. The suite consists of the following products:

- Intel® C/C++ and Fortran Compilers
- Intel® Performance Libraries
- VTune™ Performance Analyzer
- Intel® Thread Checker
- Thread Profiler

For more information on Intel software development products, please refer to the following web site: http://www.intel.com/software/products.

The Intel® Software College provides training in all Intel products as well as instruction in multithreaded programming. Please refer to the following web site for more information on the Intel Software College: https://shale.intel.com/softwarecollege.

Intel® C/C++ and Fortran Compilers

In addition to high-level code optimizations, the Intel® compilers also enable threading through automatic parallelization and OpenMP support. With automatic parallelization, the compiler detects loops that can be safely and efficiently executed in parallel and generates multithreaded code. OpenMP allows programmers to express parallelism using compiler directives and C/C++ preprocessor pragmas.

Intel® Performance Libraries

The Intel® Math Kernel Library (Intel MKL) and Intel® Integrated Performance Primitives (IPP) provide consistent performance across all Intel® microprocessors. Intel MKL provides support for BLAS, LAPACK, and vector math functions. All level-2 and level-3 BLAS functions are threaded with OpenMP. IPP is a cross-platform software library which provides a range of library functions for multimedia, audio and video codecs, signal and image processing, speech compression, and computer vision plus math support routines. IPP is optimized for Intel microprocessors and many of its component functions are already threaded with OpenMP.

VTune™ Performance Analyzer

The VTune Performance Analyzer helps developers tune their applications for optimum performance on Intel® architectures. VTune Performance tools monitor events inside Intel microprocessors to give a detailed view of application behavior, which helps identify performance bottlenecks. The VTune analyzer provides time- and event-based sampling, call-graph profiling, hotspot analysis, a tuning assistant, and many other features to assist performance tuning. It also has an integrated source viewer to link profiling data to precise locations in source code.

Intel® Thread Checker

The Intel Thread Checker facilitates debugging of multithreaded programs by automatically finding common errors such as storage conflicts, deadlock, API violations, inconsistent variable scope, thread stack overflows, etc. The non-deterministic nature of concurrency errors makes them particularly difficult to find with traditional debuggers. Thread Checker pinpoints error locations down to the source lines involved and provides stack traces showing the paths taken by the threads to reach the error. It also identifies the variables involved.

Thread Profiler

The Intel Thread Profiler facilitates tuning of OpenMP programs. It provides performance counters that are specific to OpenMP constructs. Intel Thread Profiler provides details on the time spent in serial regions, parallel regions, and critical sections and graphically displays performance bottlenecks due to load imbalance, lock contention, and parallel overhead. Performance data can be displayed for the whole program, by region, and even down to individual threads.

2.1 Automatic Parallelization With Intel® Compilers

Category

Software

Scope

Applications built with the Intel compilers for deployment on symmetric multiprocessors (SMP) and/or systems with Hyper-Threading Technology (HT).

Keywords

Auto-parallelization, data dependences, programming tools, compiler

Abstract

Multithreading an application to improve performance can be a time-consuming activity. For applications where most of the computation is carried out in simple loops, Intel compilers may be able to generate a multithreaded version automatically.

Background

The Intel C++ and Fortran compilers have the ability to analyze the dataflow in loops to determine which loops can be safely and efficiently executed in parallel. Automatic parallelization can sometimes result in shorter execution times on SMP and HT-enabled systems. It also relieves the programmer from:

- Searching for loops that are good candidates for parallel execution
- Performing dataflow analysis to verify correct parallel execution
- Adding parallel compiler directives manually.

Adding the `-Qparallel` (Windows*) or `-parallel` (Linux*) option to the compile command is the only action required of the programmer. Successful parallelization is subject to certain conditions, however, which are described in the next section.

The following Fortran program contains a loop with a high iteration count:

```
PROGRAM TEST
PARAMETER (N=100000000)
REAL A, C(N)
DO I = 1, N
   A = 2 * I – 1
   C(I) = SQRT(A)
ENDDO
PRINT*, N, C(1), C(N)
END
```

Dataflow analysis confirms that the loop does not contain data dependencies. The compiler will generate code that divides the iterations as evenly as possible among the threads at runtime. The number of threads defaults to the number of processors but can be set independently via the `OMP_NUM_THREADS` environment variable. The parallel speed-up for a given loop depends on the amount of work, the load balance among threads, the overhead of thread creation and synchronization, etc. but will, in general, be less than the

number of threads. For a whole program, speed-up depends on the ratio of parallel to serial computation (see any good textbook on parallel computing for a description of Amdahl's Law).

Advice

Three requirements must be met for the compiler to parallelize a loop. First, the number of iterations must be known before entry into a loop so that the work can be divided in advance. A while-loop, for example, usually cannot be made parallel. Second, there can be no jumps into or out of the loop. Third, and most important, the loop iterations must be independent. In other words, correct results must not logically depend on the order in which the iterations are executed. There may, however, be slight variations in the accumulated rounding error, as, for example, when the same quantities are added in a different order. In some cases, such as summing an array or other uses of temporary scalars, the compiler may be able to remove an apparent dependency by a simple transformation.

Potential aliasing of pointers or array references is another common impediment to safe parallelization. Two pointers are aliased if both point to the same memory location. The compiler may not be able to determine whether two pointers or array references point to the same memory location, for example, if they depend on function arguments, run-time data, or the results of complex calculations. If the compiler cannot prove that pointers or array references are safe and that iterations are independent, it will not parallelize the loop, except in limited cases when it is deemed worthwhile to generate alternative code paths to test explicitly for aliasing at run-time. If the programmer knows that parallelization of a particular loop is safe, and that potential aliases can be ignored, this can be communicated to the compiler with a C pragma (`#pragma parallel`) or Fortran directive (`!DIR$ PARALLEL`). An alternative way in C to assert that a pointer is not aliased is to use the `restrict` keyword in the pointer declaration, along with the `-Qrestrict` (Windows) or `-restrict` (Linux) command-line option. The compiler will never parallelize a loop that it can prove to be unsafe.

The compiler can only effectively analyze loops with a relatively simple structure. For example, it cannot determine the thread safety of a loop containing external function calls because it does not know whether the function call has side effects that introduce dependences. Fortran 90 programmers can use the `PURE` attribute to assert that subroutines and functions contain no side effects. Another way, in C or Fortran, is to invoke inter-procedural optimization with the `-Qipo` (Windows) or `-ipo` (Linux) compiler option. This gives the compiler the opportunity to analyze the called function for side effects.

When the compiler is unable to parallelize automatically loops that the programmer knows to be parallel, OpenMP should be used. In general, OpenMP is the preferred solution because the programmer typically understands the code better than the compiler and can express parallelism at a coarser granularity (see Application Threading, 3.2: Granularity And Parallel Performance). On the other hand, automatic parallelization can be effective for nested loops, such as those in a matrix multiply. Moderately coarse-grained parallelism results from threading of the outer loop, allowing the inner loops to be optimized for fine-grained parallelism using vectorization or software pipelining.

Just because a loop *can* be parallelized does not mean that it *should* be parallelized. The compiler uses a threshold parameter to decide whether to parallelize a loop. The `-Qpar_threshold[n]` (Windows) and `-par_threshold[n]` (Linux) compiler options adjust this parameter. The value of `n` ranges from 0 to 100, where 0 means to always parallelize a safe loop and 100 tells the compiler to only parallelize those loops for which a performance gain is highly probable. The default value of `n` is 75.

The switches `-Qpar_report[n]` (Windows) or `-par_report[n]` (Linux), where `n` is 1 to 3, can be used to learn which loops were parallelized. Look for messages such as:

```
test.f90(6) : (col. 0) remark:  LOOP WAS AUTO-PARALLELIZED
```

The compiler will also report which loops could not be parallelized and the reason why, e.g.:

```
serial loop: line 6
flow data dependence from line 7 to line 8, due to “c”
```

This is illustrated by the following example:

```
void add (int k, float *a, float *b)
{
   for (int i = 1; i < 10000; i++)
      a[i] = a[i+k] + b[i];
}
```

The compile command ‘`icl -c -Qparallel -Qpar_report3 add.cpp`’ results in the following messages:

```
add.cpp
   procedure: add
   serial loop: line 2
      anti data dependence assumed from line 2 to line 2, due to "a"
      flow data dependence assumed from line 2 to line 2, due to "a"
      flow data dependence assumed from line 2 to line 2, due to "a"
```

Because the compiler does not know the value of `k`, it must assume that the iterations depend on each other, as for example if `k` equals -1. The programmer may know otherwise, however, due to specific knowledge of the application (e.g., `k` always greater than 10000), and can override the compiler by inserting a pragma:

```
void add (int k, float *a, float *b)
{
   #pragma parallel
      for (int i = 1; i < 10000; i++)
         a[i] = a[i+k] + b[i];
}
```

The messages now show that the loop is parallelized:

```
add.cpp
add.cpp(3) : (col. 3) remark: LOOP WAS AUTO-PARALLELIZED.
   procedure: add
   parallel loop: line 3
      shared: {"b", "a", "k"}
      private: {"i"}
      first private: { }
      reductions: { }
```

However, it is now the programmer's responsibility not to call this function with a value of `k` that is less than 10000, as this could lead to incorrect results.

Usage Guidelines

Try building the computationally intensive kernel of your application with the `-parallel` (Linux) or `-Qparallel` (Windows) compiler switch. Enable reporting with `-par_report3` (Linux) or `-Qpar_report3` (Windows) to find out which loops were parallelized and which loops could not be parallelized. For the latter, try to remove data dependencies and/or help the compiler disambiguate potentially aliased memory references.

The transformations necessary to parallelize a loop may sometimes impact other high-level optimizations (e.g., loop inversion). This can often be recognized from the compiler optimization reports. Always measure performance with and without parallelization to verify that a useful speedup is being achieved.

If `-openmp` and `-parallel` are both specified on the same command line, the compiler will only attempt to parallelize those functions that do not contain OpenMP directives.

For builds with separate compiling and linking steps, be sure to link the OpenMP runtime library when using automatic parallelization. The easiest way to do this is to use the compiler driver for linking, e.g.: `icl -Qparallel` (IA-32 Windows) or `efc -parallel` (Itanium processor for Linux).

References

In this series, see also:

This section, 2.2: Multithreaded Functions In The Intel® Math Kernel Library

This section, 2.4: Using Thread Profiler To Evaluate OpenMP Performance

Application Threading, 3.2: Granularity And Parallel Performance

Synchronization, 3.5: Expose Parallelism By Avoiding Or Removing Artificial Dependencies

See also:

The Intel® C++ Compiler User's Guide or *The Intel® Fortran Compiler User's Guide*, see "Compiler Optimizations/Parallelization/Automatic Parallelization"

"Efficient Exploitation of Parallelism on Pentium® III and Pentium 4 Processor-Based Systems", Aart Bik, Milind Girkar, Paul Grey and Xinmin Tian, *Intel Technology Journal*

http://www.intel.com/technology/itj/q12001/articles/art_6.htm

The Intel Software College provides extensive training material on Intel software development products.

2.2 Multithreaded Functions In The Intel® Math Kernel Library

Category

Software

Scope

Applicable to 32-bit processors from the Pentium processor through the Intel® Xeon™ processor and to the Intel® Itanium® processor family on both the Windows* and Linux* operating systems

Keywords

Math Kernel Library, BLAS, LAPACK, FFT, programming tools

Abstract

A number of key and appropriate routines within the Intel® Math Kernel Library (Intel MKL) have been threaded to provide increased performance on systems with multiple processors in a shared-memory environment. We will show that the use of this library makes available to the user an easy way to get high performance on key algorithms, both on single processor systems and on multiprocessor systems. The user need only tell the system how many processors to use.

Background

A great deal of scientific code can be parallelized, but not all of it will run faster on multiple processors on an SMP system because there is inadequate memory bandwidth to support the operations. Fortunately, important elements of technical computation in finance, engineering and science rely on arithmetic operations that can effectively use cache, which reduces the demands on the memory system. The basic condition that must be met in order for multiple processors to be effectively used on a task is that the reuse of data in cache must be high enough to free the memory bus for the other processors. Operations such as factorization of dense matrices and matrix multiplication (a key element in factorization) can meet this condition if the operations are structured properly.

It may be possible to get a substantial percentage of peak performance on a processor simply by compiling the code, possibly along with some high-level code optimizations. However, if the resulting code relies heavily on memory bandwidth, then it probably will not scale well when the code is parallelized because, it will not scale well because there will be inadequate cache usage, and with that, inadequate memory bandwidth to supply all the processors.

Widely used functions such as the level-3 BLAS (basic linear algebra subroutines) (all matrix-matrix operations), many of the LAPACK (linear algebra package) functions, and, to a lesser degree, DFT's (discrete Fourier transforms) all can reuse data in cache sufficiently that multiple processors can be supported on the memory bus.

Advice

There are really two parts to the advice. First, wherever possible the user should employ the widely used, *de facto* standard functions from BLAS and LAPACK since these are available in source code form (the user can build them) and many hardware vendors supply optimized versions of these functions for their machines. Just linking to the high-

performance library may improve the performance of an application substantially, depending on the degree to which the application depends on LAPACK, and by implication, the BLAS (since LAPACK is built on the BLAS).

Intel MKL is Intel's library containing these functions. The level-3 BLAS have been tuned for high performance on a single processor but have also been threaded to run on multiple processors and to give good scaling when more than one processor is used. Key functions of LAPACK have also been threaded. Good performance on multiple processors is possible just with the threaded BLAS but threading LAPACK improves performance for smaller-sized problems. The LINPACK benchmark, which solves a set of equations, demonstrates well the kind of scaling that threading of these functions can yield. This benchmark employs two high-level functions from LAPACK – a factorization and a solving routine. Most of the time is spent in the factorization. For the largest test problem, Intel MKL achieved a 3.84 speedup on four processors, or 96% parallel efficiency.

In addition to these threaded routines, the DFT's are also threaded and scale very well. For example, on 1280x1280 single precision complex 2D transforms, the performance on the Itanium 2 processor for one, two, and four processors is 1908, 3225 (1.69 speedup), and 7183 MFLOPS (3.76 speedup), respectively

Usage Guidelines

There are caveats in the use of these functions with the current releases of Intel MKL (up through version 6.0 beta update) that have nothing to do with the library directly. Problems can arise depending on the environment.

OpenMP is used to thread Intel MKL, and Intel MKL uses the same OpenMP runtime library as the Intel compilers. Therefore, problems can arise when OpenMP applications that use Intel MKL are not compiled with the Intel compilers. Specifically, the application will attempt to use two different OpenMP libraries, one from the non-Intel compiler and the other from Intel MKL. When the `OMP_NUM_THREADS` environment variable is greater than one, chaos results when both libraries attempt to create threads, and the program will fail. A future version of Intel MKL will provide an alternate means of controlling thread creation. In the meantime, if this problem is encountered, the issue should be submitted to Intel through Intel Premier support (http://premier.intel.com) for an interim solution.

A second issue can arise on clusters with symmetric multiprocessor nodes[1]. MPI or PVM applications running on such clusters often create one process for each processor in a node. A node is defined as a computer with an operating system image. In a typical cluster, an operating system is installed on each computer in the cluster. If these applications use Intel MKL, threads may also be created by each MPI or PVM process. This could result in over-subscription of processor resources within a node. For MPI or PVM applications that create one process per processor, it is recommended that `OMP_NUM_THREADS` be set to one.

---

[1] A node is defined as a computer with an operating system image. In a typical cluster, an operating system is installed on each computer in the cluster.

References

In this chapter, see also:

2.1: Automatic Parallelization With Intel® Compilers

2.5: Using Thread Profiler To Evaluate OpenMP Performance

See also:

The Intel Math Kernel Library can be obtained at http://developer.intel.com/software/products/perflib/.

The Intel Software College provides extensive training material on Intel software development products.

Information about the BLAS and LAPACK can be obtained at http://www.netlib.org.

2.3 Avoiding And Identifying False Sharing Among Threads With The VTune™ Performance Analyzer

Category

Software

Scope

General multithreading

Keywords

VTune, cache coherence, data alignment, profiler, programming tools

Abstract

In symmetric multiprocessor (SMP) systems, each processor has a local cache. The memory system must guarantee cache coherence. False sharing occurs when threads on different processors modify different variables that reside on the same cache line. Each write will invalidate the line in other caches, forcing an update and hurting performance. This topic covers methods to detect and correct false sharing using the Intel VTunePerformance Analyzer.

Background

False sharing is a well-known performance issue on SMP where each processor has a local cache. It occurs when threads on different processors modify variables that reside on the same cache line, as illustrated in Figure 1. The reason this is called false sharing is that each thread is not actually sharing access to the same variable. Access to the same variable, or true sharing, would require programmatic synchronization constructs to ensure ordered data access.

The source line highlighted in red in the following example code causes false sharing:

```
double sum=0.0, sum_local[NUM_THREADS];

#pragma omp parallel num_threads(NUM_THREADS)
{
   int me = omp_get_thread_num();
   sum_local[me] = 0.0;

   #pragma omp for
   for (i = 0; i < N; i++)
      sum_local[me] += x[i] * y[i];

   #pragma omp atomic
   sum += sum_local[me];
}
```

There is a potential for false sharing on array `sum_local`. This array is dimensioned according to the number of threads and is small enough to fit in a single cache line. When executed in parallel, the threads modify different, but adjacent, elements of `sum_local` (the source line highlighted in red), which invalidates the cache line for all processors.

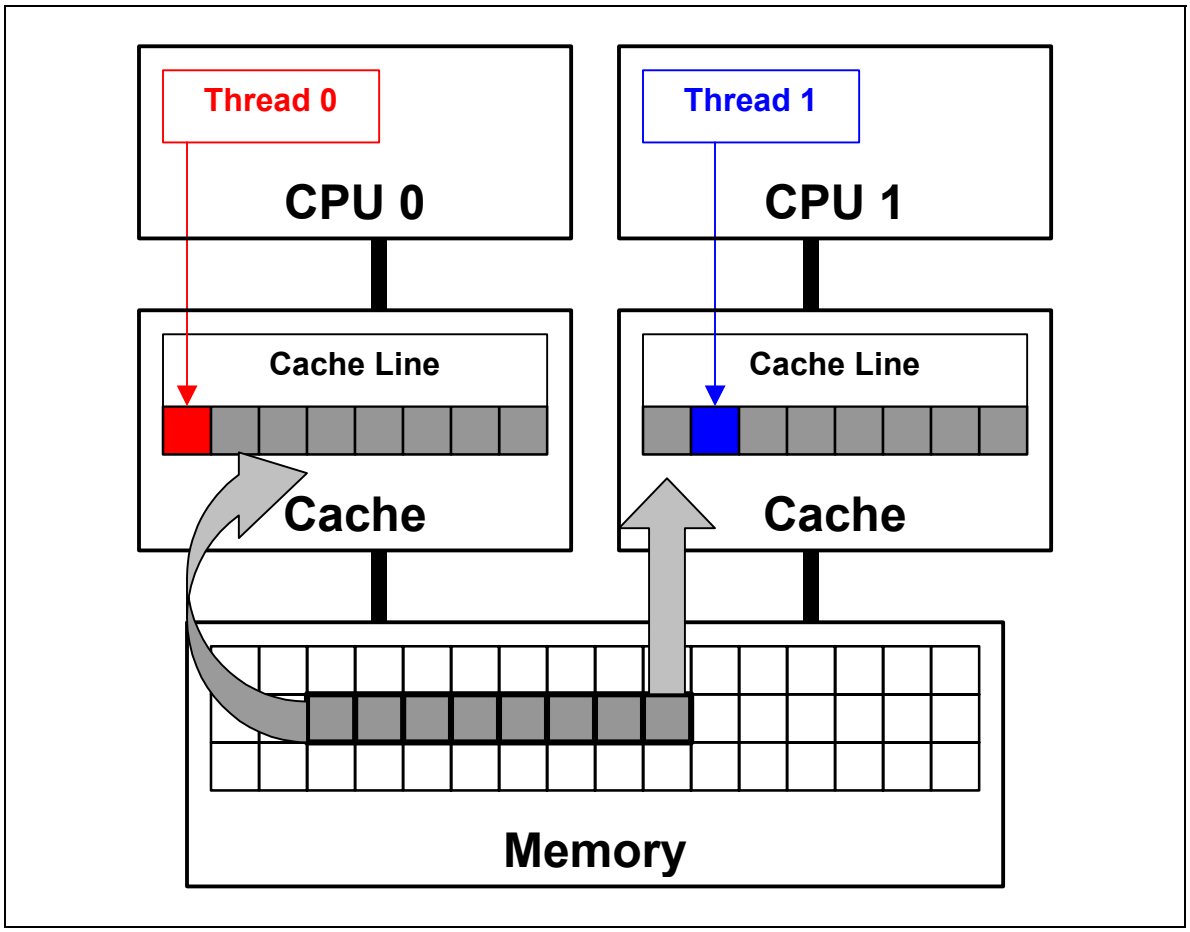

Figure 1. False sharing of a cache line

False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces a memory update to maintain cache coherency. This is illustrated in the diagram (top). Threads 0 and 1 require variables that are adjacent in memory and reside on the same cache line. The cache line is loaded into the caches of CPU 0 and CPU 1 (gray arrows). Even though the threads modify different variables (red and blue arrows), the cache line is invalidated. This forces a memory update to maintain cache coherency.

To ensure data consistency across multiple caches, Intel multiprocessor-capable processors follow the MESI (Modified/Exclusive/Shared/Invalid) protocol. On first load of a cache line, the processor will mark the cache line as 'Exclusive' access. As long as the cache line is marked exclusive, subsequent loads are free to use the existing data in cache. If the processor sees the same cache line loaded by another processor on the bus, it marks the cache line with 'Shared' access. If the processor stores a cache line marked as 'S', the cache line is marked as 'Modified' and all other processors are sent an 'Invalid' cache line message. If the processor sees the same cache line which is now marked 'M' being accessed by another processor, the processor stores the cache line back to memory and marks its cache line as 'Shared'. The other processor that is accessing the same cache line incurs a cache miss.

The frequent coordination required between processors when cache lines are marked 'Invalid' require cache lines to be written to memory and subsequently loaded. False

sharing increases this coordination and can significantly degrade application performance.

Advice

The basic advice of this section is to avoid false sharing in multithreaded applications. Detecting false sharing when it is already present is another matter, however. The first method of detection is through code inspection. Look for instances where threads access global or dynamically allocated shared data structures. These are potential sources of false sharing. Note that false sharing can be obscure in that threads may be accessing completely different global variables that just happen to be relatively close together in memory. Thread-local storage or local variables can be ruled out as sources of false sharing.

A better detection method is to use the Intel VTune Performance Analyzer. For multiprocessor systems, configure VTune analyzer to sample the '`2nd Level Cache Load Misses Retired`' event. For Hyper-Threading Technology-enabled processors, configure VTune analyzer to sample the 'Memory Order Machine Clear' event. If you have a high occurrence and concentration of these events at or near load/store instructions within threads, you likely have false sharing. Inspect the code to determine the likelihood that the memory locations reside on the same cache line.

Once detected, there are several techniques to correct false sharing. The goal is to ensure that variables causing false sharing are spaced far enough apart in memory that they cannot reside on the same cache line. Not all possible techniques are discussed here. Below are three possible methods.

One technique is to use the Intel Compiler 7.0 or Microsoft Visual Studio* .NET directives to force individual variable alignment. The following source code demonstrates the compiler technique using '`__declspec (align(n))`' where `n` equals 16 (128 byte boundary) to align the individual variables on cache line boundaries. Note that not all compilers support 128 byte alignment using this technique.

```
__declspec (align(128)) int thread1_global_variable;
__declspec (align(128)) int thread2_global_variable;
```

When using an array of data structures, pad the structure to the end of a cache line to ensure that the array elements begin on a cache line boundary. If you cannot ensure that the array is aligned on a cache line boundary, pad the data structure to twice the size of a cache line. The following source code demonstrates padding a data structure to a cache line boundary and ensuring that the array is also aligned using the compiler '`__declspec (align(n))`' statement where `n` equals 16 (128 byte boundary). If the array is dynamically allocated, you can increase the allocation size and adjust the pointer to align

with a cache line boundary.

```
struct ThreadParams
{
      // For the following 4 variables: 4*4 = 16 bytes
      unsigned long thread_id;
      unsigned long v;   // Frequent read/write access variable
      unsigned long start;
      unsigned long end;
   // expand to 128 bytes to avoid false-sharing
   // (4 unsigned long variables + 28 padding)*4 = 128
      int padding[28];
};

__declspec (align(128)) struct ThreadParams Array[10];
```

It is also possible to reduce the frequency of false sharing by using thread-local copies of data. The thread-local copy can be read and modified frequently, and the result copied back to the data structure only when complete. The following source code demonstrates using a local copy to avoid false sharing.

```
struct ThreadParams
{
   // For the following 4 variables: 4*4 = 16 bytes
   unsigned long thread_id;
   unsigned long v;   //Frequent read/write access variable
   unsigned long start;
   unsigned long end;
};

void threadFunc(void *parameter)
{
   ThreadParams *p = (ThreadParams*) parameter;
   // local copy for read/write access variable
   unsigned long local_v = p->v;

   for(local_v = p->start; local_v < p->end; local_v++)
   {
      // Functional computation
   }
   p->v = local_v; // Update shared data structure only once
}
```

Usage Guidelines

Avoid false sharing but use these techniques sparingly. Overuse of these techniques, where they are not needed, can hinder the effective use of the processor's available cache.

Even with multiprocessor shared-cache designs, it is recommended that you avoid false sharing. The small potential gain for trying to maximize cache utilization on multiprocessor shared cache designs does not generally outweigh the software maintenance costs required to support multiple code paths for different cache architectures.

References

In this series, see also:

This chapter, 2.5: Using Thread Profiler To Evaluate OpenMP Performance

Memory Management, 5.3: Offset Thread Stacks To Avoid Cache Conflicts On Intel® Processors With Hyper-Threading Technology

See also:

The Intel Software College provides extensive training material on Intel software development products. The online course “Getting Started with the VTune Performance Analyzer” is recommended with respect to the present topic.

2.4 Find Multithreading Errors With The Intel Thread Checker

Category

Software

Scope

Automated debugging of multithreaded applications in the Win32 environment

Keywords

Thread Checker, VTune, debugger, programming tools, race conditions

Abstract

The Intel® Thread Checker, one of the Intel® Threading Tools, is used to debug multithreading errors in Win32 applications. It supports OpenMP and the Win32 multithreading API. Thread Checker automatically finds storage conflicts, deadlock, or conditions that could lead to deadlock, thread stalls, abandoned locks, and more.

Background

Multithreaded programs have temporal component that makes them more difficult to debug than serial programs. Concurrency errors (e.g., data races anddeadlock) are difficult to find and reproduce because they are non-deterministic. If the programmer is lucky, the error will always crash or deadlock the program. If the programmer is not so lucky, the program will execute correctly 99% of the time, or the error will result in slight numerical drift that only becomes apparent after long execution times.

Traditional debugging methods are poorly suited to multithreaded programs. Debugging probes (i.e., print statements) often mask errors by changing the timing of multithreading programs. Executing a multithreaded program inside a debugger can give some information, provided that the bugs can be consistently reproduced. However, the programmer must sift through multiple thread states (i.e., instruction pointer, stack) to diagnose the error.

The Intel Thread Checker is designed specifically for debugging multithreaded programs. It finds the most common concurrent programming errors and pinpoints their locations in the program, such as the following.:

- Storage conflicts – The most common concurrency error involves unsynchronized modification of shared data. For example, multiple threads simultaneously incrementing the same static variable can result in data loss but are not likely to crash the program. The next section shows how to use the Intel Thread Checker to find such errors.
- Deadlock – When a thread must wait for a resource or event that will never occur, it is deadlocked. Bad locking hierarchies are a common cause. For example, a thread tries to acquire locks A and B, in that order, while another thread tries to acquire the locks in the reverse order. Sometimes the code executes without deadlock (

Table 2.1).

Table 2.1. A bad locking hierarchy can sometimes execute without deadlock

Time	Thread 1	Thread 2
T0	Acquire lock A	
T1	Acquire lock B	
T2	Perform task	
T3	Release lock B	
T4	Release lock A	
T5		Acquire lock A
T6		Acquire lock B
T7		Perform task
T8		Release lock B
T9		Release lock A

This locking hierarchy can also deadlock both threads (Table 2.2), however. Both threads are waiting for resources that they can never acquire. Thread Checker identifies deadlock and the potential for deadlock, as well as the contested resources.

Table 2.2. Deadlock due to a bad locking hierarchy

Time	Thread 1	Thread 2
T0	Acquire lock A	
T1		Acquire lock B
T2		Wait for lock A
T3	Wait for lock B	

- Abandoned locks – Thread Checker detects when a thread terminates while holding a Win32 critical section or mutex variable because this can lead to deadlock or unexpected behavior. Threads waiting on an abandoned critical section are deadlocked. Abandoned mutexes are reset.
- Lost signals – Thread Checker detects when a Win32 event variable is pulsed (i.e., the Win32 `PulseEvent` function) when no threads are waiting on that event, because this is a common symptom of deadlock. For example, the programmer expects a thread to

be waiting before an event is pulsed. If the event is pulsed before the thread arrives, the thread may wait for a signal that will never come.

Thread Checker also finds many other types of errors, including API usage violations, thread stack overflows, and scope violations.

Advice

Use the Intel Thread Checker to facilitate debugging of OpenMP and Win32 multithreaded applications. Errors in multithreaded programs are harder to find than errors in serial programs not only because of the temporal component mentioned above, but also because such errors are not restricted to a single location. Threads operating in distant parts of the program can cause errors. Thread Checker can save an enormous amount of debugging time, as illustrated by the simple example shown below.

To prepare a program for Thread Checker analysis, compile with optimization *disabled* and debugging symbols *enabled*. Link the program with the `/fixed:no` option so that the executable can be relocated. Thread Checker instruments the resulting executable image when it is run under the VTune Performance Analyzer, Intel's performance tuning environment. For *binary* instrumentation, either the Microsoft Visual C++ compiler (version 6.0) or the Intel C++ and Fortran compilers (version 7.0 or later) may be used. The Intel compilers support *source-level* instrumentation (the `/Qtcheck` option), however, which provides more detailed information.

The following program contains a subtle race condition:

The program is supposed to create four threads that report their identification numbers. Sometimes the program gives the expected output:

```
Thread 0 reporting
Thread 1 reporting
Thread 2 reporting
Thread 3 reporting
```

Threads do not always report in the order of their identification numbers but all threads print a message. Other times, some threads appear to report more than once, others do not report at all, and a mysterious new thread appears, e.g.:

```
Thread 2 reporting
Thread 3 reporting
Thread 3 reporting
Thread 4 reporting
```

Thread Checker easily finds the error in this program and shows the statements responsible (Figure 2):

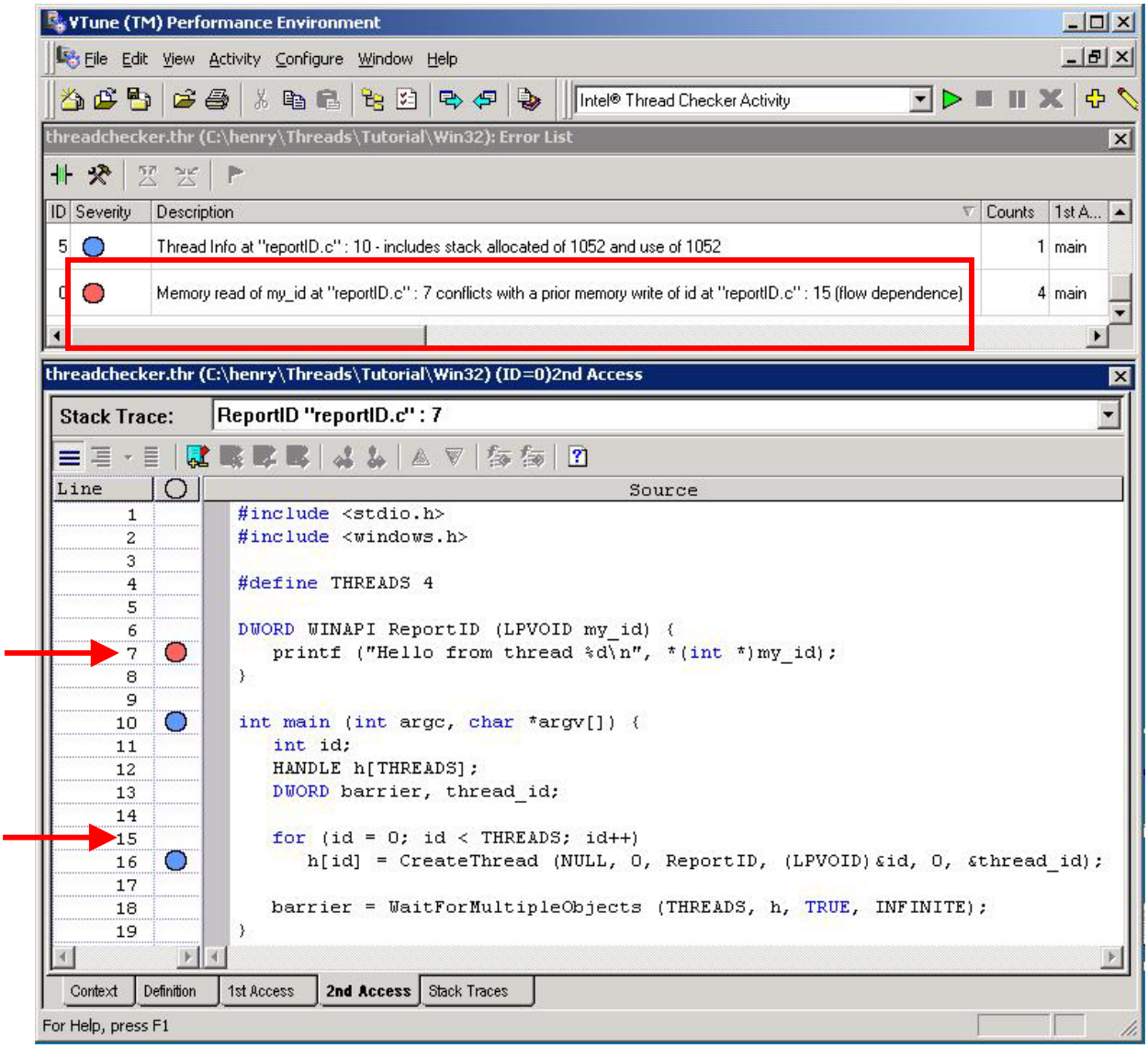

Figure 2. Intel Thread Checker

The error description (see the red box) explains the storage conflict in plain English – a thread is reading variable my_id on line-7 while another thread is simultaneously writing variable id on line-15. The variable my_id in function ReportID is a pointer to variable id, which is changing in the main routine. The programmer mistakenly assumes that a thread begins executing the moment it is created. However, the operating system may schedule threads in any order. The main thread can create all worker threads before any of them begin executing. Correct this error by passing each thread a pointer to a unique location that is not changing.

Usage Guidelines

Intel Thread Checker currently is available for the 32-bit versions of the Microsoft Windows 2000* and Windows XP* operating systems. Thread Checker supports both OpenMP and the Win32 threading API. The Intel compilers are required for OpenMP support. They are also required for more detailed source-level instrumentation. Otherwise, the Microsoft Visual C++ compiler may be used.

Note that the Intel Thread Checker performs dynamic analysis, not static analysis. Thread Checker only analyzes code that is executed. Therefore, multiple analyses exercising different parts of the program may be necessary to ensure adequate code coverage.

Thread Checker instrumentation increases the CPU and memory requirements of an application so choosing a small but representative test problem is very important. Workloads with runtimes of a few seconds are best. Workloads do not have to be realistic. They just have to exercise the relevant sections of multithreaded code. For example, when debugging an image processing application, a 10 x 10 pixel image is sufficient for Thread Checker analysis. A larger image would take significantly longer to analyze but would not yield additional information. Similarly, when debugging a multithreaded loop, reduce the number of iterations.

References

The Intel Thread Checker web site

"Getting Started with the Intel Threading Tools," distributed with Intel Threading Tools.

"Intel Thread Checker Lab," distributed with the Intel Threading Tools.

The Intel Software College provides extensive training material on Intel software development products. The online course "Using the Intel Threading Tools" is recommended with respect to the present topic.

2.5 Using Thread Profiler To Evaluate OpenMP Performance

Category

Software

Scope

OpenMP performance tuning on Windows platforms

Keywords

Profiler, programming tools, OpenMP, VTune, parallel overhead

Abstract

Thread Profiler is one of the Intel Threading Tools. It is used to evaluate performance of OpenMP threaded codes, to identify performance bottlenecks, and to gauge scalability of OpenMP applications.

Background

Once an application has been debugged and is running correctly, engineers often turn to performance tuning. Traditional profilers are of limited use in tuning OpenMP for a variety of reasons (unaware of OpenMP constructs, cannot report load imbalance, do not report contention for synchronization objects).

Thread Profiler is designed to understand OpenMP threading constructs and to measure their performance over the whole application run, within each OpenMP region, and down to individual threads. Thread Profiler is able to detect and measure load imbalance (from uneven amounts of computation assigned to threads), time spent waiting for synchronization objects as well as time spent in critical regions, time spent at barriers, and time spent in the Intel OpenMP Runtime Engine (parallel overhead).

Advice

To prepare an OpenMP application for use with the Thread Profiler, build an executable that includes the OpenMP profiling library (use `/Qopenmp_profile` compiler switch). When setting up a Thread Profiler Activity in VTune Performance Analyzer, be sure to use a full, production data set running with an appropriate number of threads. Best results for production performance tuning will be obtained using a representative data set that exercises the code as close to normal as possible. Small, test data sets may not fully exercise the parallelism of the code or the interaction between threads, which can lead to overlooking serious performance problems. While the execution time will be increased by the instrumentation of the OpenMP threads, this increase is minimal.

Once the application has completed execution, summary performance results are displayed in the Thread Profiler window. There are three graphical views of the performance data that can be used. Each is accessible from separate tabs found below the Legend pane. These three views are summarized below:

- **Summary View** – This view is the default for the Thread Profiler (Figure 3). The histogram bar is divided into a number of regions indicating the average amount of time the application spent in the observed performance category. These performance categories are as follows:

- parallel execution (time within OpenMP parallel regions) in green,
- sequential time in blue,
- idle time due to load imbalance between threads in red,
- idle time waiting at barriers in purple,
- idle time spent waiting to gain access to synchronization objects in orange,
- time spent executing within critical regions in gray, and
- parallel (time spent in OpenMP Runtime Engine) and sequential (time spent in OpenMP regions that are not executed in parallel) overheads in yellow and olive, respectively.

Left-clicking on the bar will populate the legend with numerical details about total execution time for each category over the entire run of the application.

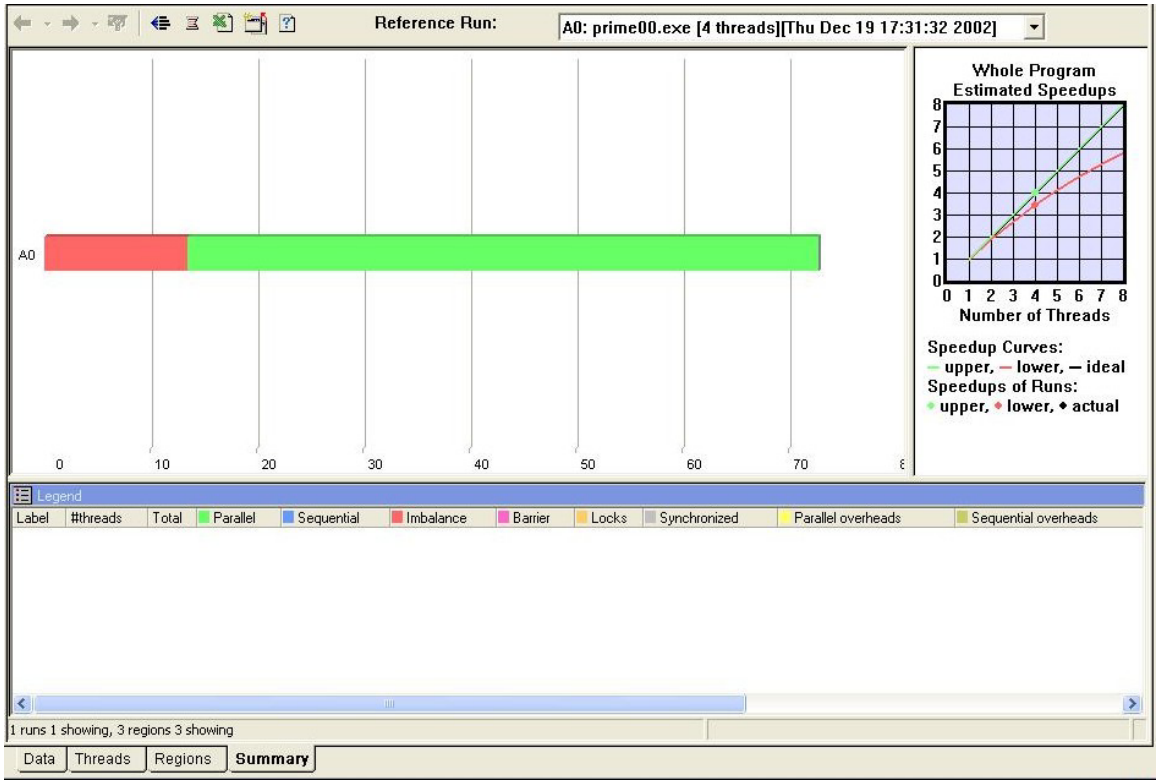

Figure 3. Summary View in Thread Profiler

Of course, the best possible display is a histogram that is mostly green with minimal amounts of blue sequential time. Large amounts of other colors within the summary histogram are an indication of performance problems. The severity of any problems noted will depend on the type of problem cited and the actual amount of time spent in that category. Relatively small performance problems may be tolerable, especially if it is determined that no easy fix would be possible due to algorithmic implementation.

The Summary View can also be used to compare scalability of an application with varying numbers of threads. Just drag and drop different activity runs of the same code with the same data executed with different numbers of threads onto the Summary View. Besides showing scalability, some performance obstacles may manifest themselves as the number of threads is varied. For example, lock contention often increases as more threads are added, which can prevent some applications from scaling well even when adequate resources are available.

After deciding to pursue a performance problem seen in the Summary View, a more detailed analysis will need to be done in order to locate and identify the source of the problem. Examining the timing data through the Regions View does this.

- **Regions View:** This view breaks down the summary data by each region within the source code (Figure 4). These include the OpenMP parallel regions and the surrounding sequential regions. The Regions View gives you the power to determine which parts of the code are causing the performance problems, whether it is one single region or all regions. Observation of large sequential regions could be used to identify portions of the code for further parallel development. Click on regional histograms in order to populate the Legend pane with numerical details about the time spent within each performance category. Multiple regions can be selected and compared in the Legend.

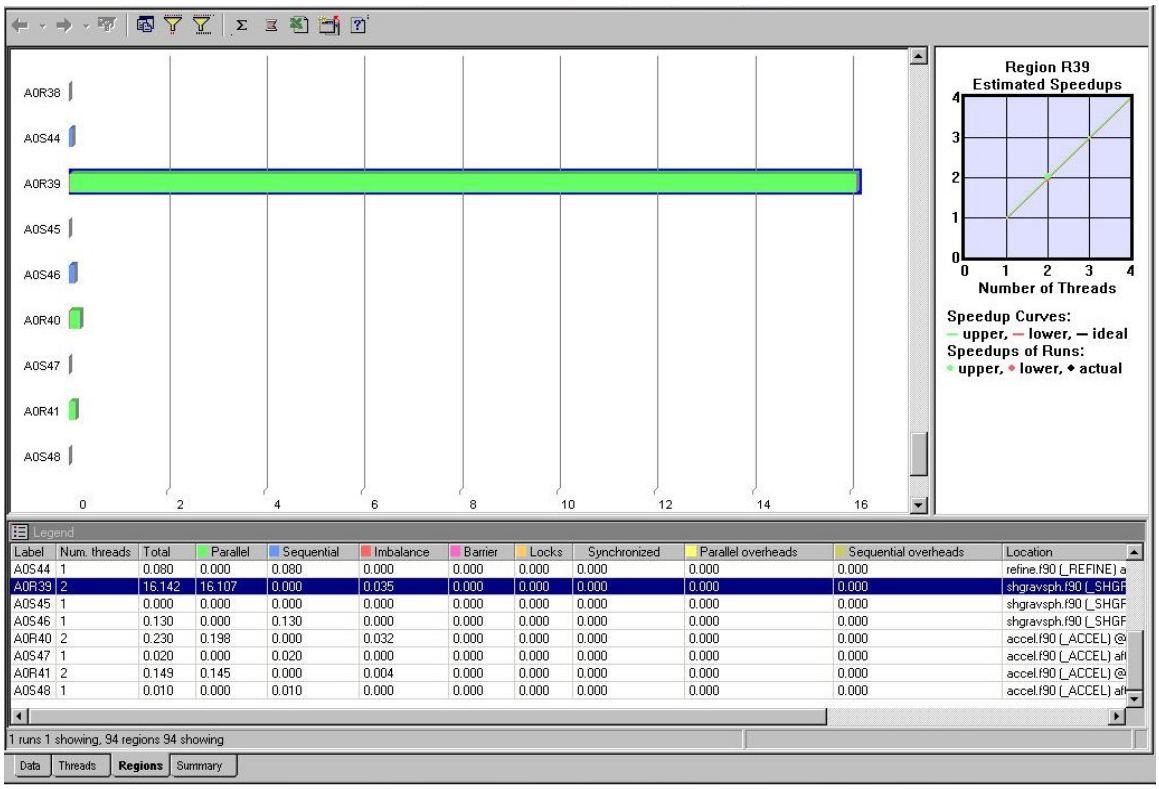

Figure 4. Regions View and Legend in Thread Profiler

The Figure 4 shows a set of parallel and serial regions from an application. This view contains one parallel region (A0R39) that accounts for much of the time spent in the application, several smaller parallel regions, and several sequential regions. The sequential regions shown are too small to consider further parallelization.

Right-clicking on a selected region histogram (surrounded with blue outline) pops up a menu dialog that includes an option to display source code. Thus, once you've determined a region that you wish to tune, you can find the corresponding source code for assessment of the cause and devise a solution. The source code locations for regions are also noted in the Legend pane.

- **Threads View:** The Threads View gives a more detailed presentation of timing characteristics of the application (Figure 5). A separate histogram will be present for each thread that was used in the execution. The data will, by default, be summary data for the entire run broken down to the performance of each thread. The master thread will be the only one with sequential time; all other thread histograms will be shorter by this sequential time.

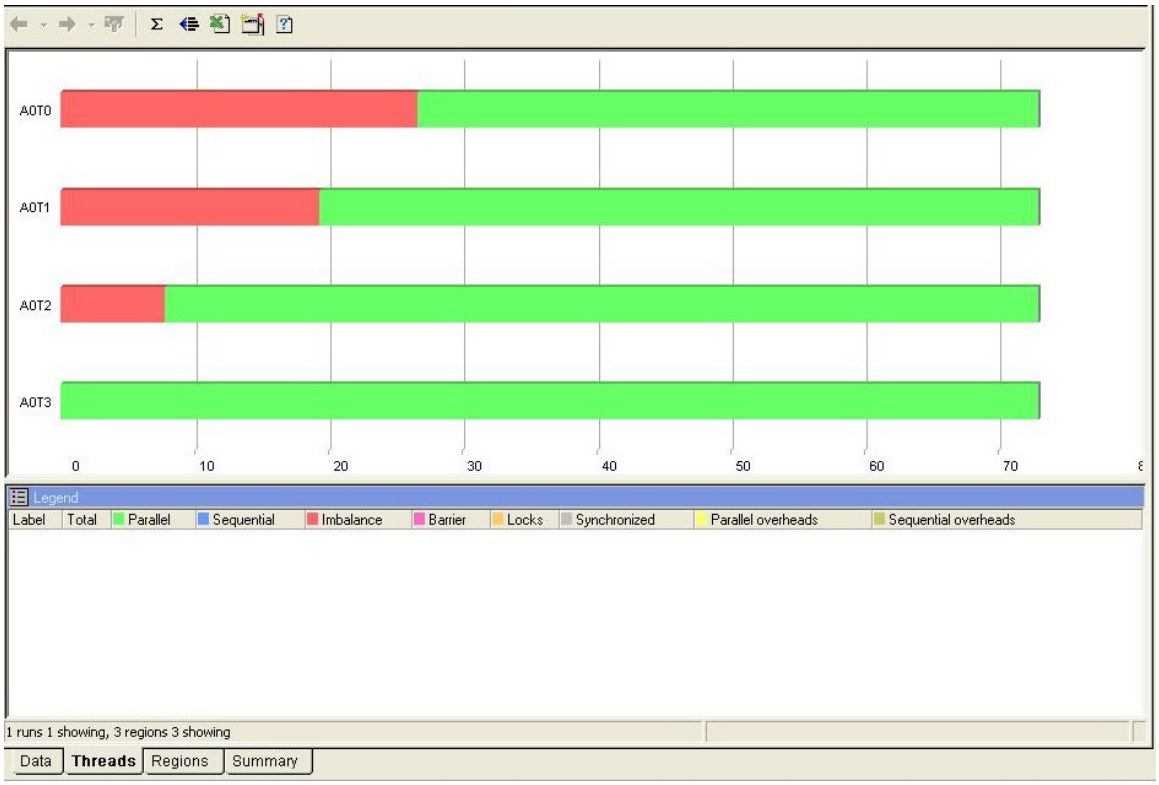

Figure 5. Threads View in Thread Profiler

By starting with the Regions view and first filtering out all but the relevant region(s) of interest, the Threads View can then be used to focus on individual thread performance within specific regions. This level of detail can give more clues as to the cause of performance problems. For instance, do all threads exhibit roughly the same amount of performance overhead? Is the performance overhead only exhibited within a single thread? Or is there some other pattern of performance?

In Figure 5, the Threads View has been filtered down to a single parallel region. You can see a "stair step" of load imbalance across the four threads used in the region. This performance relationship indicates a regular pattern of increasing computation over loop iterations. That is, successive loop iterations require more processing time than previous iterations. OpenMP uses static scheduling by default. Since the rise in computation time between iterations is fairly constant, static scheduling with a small chunk size will achieve good load balance and fix the performance bottleneck. If the variation of work for each loop iteration were less predictable, dynamic scheduling of the iterations would be more appropriate.

Usage Guidelines

The Thread Profiler currently only supports OpenMP threaded codes run on Microsoft Windows operating systems. The Intel® 7.0 compilers are needed to be able to compile for OpenMP threading and to have the OpenMP profiling library available.

References

In this series, see also:

This chapter, 2.3: Avoiding And Identifying False Sharing Among Threads With The VTune™ Performance Analyzer

Application Threading, 3.2: Granularity And Parallel Performance

Application Threading, 3.3: Load Balance And Parallel Performance

Application Threading, 3.8: Exploiting Data Parallelism In Ordered Data Streams

Application Threading, 3.9: Manipulate Loop Parameters To Optimize OpenMP Performance

Synchronization, 4.1: Managing Lock Contention, Large And Small Critical Sections

Memory Management, 5.2: Use Thread-Local Storage To Reduce Synchronization

See also:

The Thread Profiler web site

"Getting Started with the Intel Threading Tools," distributed with Intel Threading Tools.

The Intel Software College provides extensive training material on Intel software development products. The online course "Using the Intel Threading Tools" is recommended with respect to the present topic.

3 Application Threading

This chapter covers general topics in application threading, particularly with respect to parallel performance. The topics occasionally refer to API-specific issues but much of the advice applies to any parallel programming method.

The chapter begins with a discussion of data vs. functional decomposition. The opening topic gives advice on choosing the most appropriate threading method for either parallel model. This is followed by topics on granularity and load balance. These are critical issues in parallel programming because they directly affect the efficiency and scalability of a multithreaded application.

Tailoring thread behavior to a particular runtime environment is often overlooked in multithreaded programs. On a single-user system, for example, allowing idle threads to spin may be more efficient than putting them to sleep. On shared systems, however, forcing idle threads to yield the CPU may be more efficient. The issues involved in threading for high turnaround vs. high throughput are discussed.

Many algorithms contain optimizations that benefit serial performance but inadvertently introduce dependencies that inhibit parallelism. It is often possible to remove such dependencies through simple transformations. Techniques for exposing parallelism by avoiding or removing artificial dependencies are discussed.

The next two topics describe how to choose an appropriate number of threads and how to minimize overhead due to thread creation. Creating too many threads hurts performance for many reasons, including increased system overhead, decreased granularity, increased lock contention, etc. Therefore, it is a good idea to control the number of threads through runtime heuristics and thread pools. Heuristics allow the programmer to create threads based on workload requirements that may not be known until runtime. Use of thread pools to limit the overhead of thread creation is described. The advice in this topic is primarily for applications threaded with Pthreads or the Win32* thread API. Thread pools are already used in the Intel® OpenMP* implementation.

The chapter closes with techniques for handling order-dependent output and loop optimizations designed to boost OpenMP performance.

3.1 Choosing An Appropriate Threading Method: OpenMP Versus Explicit Threading

Category

Application Threading

Scope

General multithreading

Keywords

OpenMP, POSIX threads, Pthreads, Win32 threads, data parallelism, functional decomposition

Abstract

Of the two most common approaches to multithreading, compiler-based and library-based methods, neither is appropriate to all situations. Compiler-based threading methods like OpenMP are best suited to data parallelism. Methods based on threading libraries, primarily the Win32 and POSIX* thread API's, are best suited to functional decomposition.

Background

Programmers have used threads for many years to express the natural concurrency of their applications. For example, threads allow an application to continue processing while still receiving GUI input. Thus, the application is not frozen from the user's perspective. On a symmetric multiprocessor and/or CPU with Hyper-Threading Technology, threads can significantly improve performance through parallel computing.

Broadly speaking, two threading methods are available (i.e., library-based and compiler-directed), each suited to a particular type of multithreaded programming. Library-based threading methods (the Win32 multithreading API on Windows and the Pthreads library on Linux*) require the programmer to manually map concurrent tasks to threads. There is no explicit parent-child relationship between the threads – all threads are peers. This makes the threading model very general. The libraries also give the programmer control over low-level aspects of thread creation, management, and synchronization. This flexibility is the key advantage of library-based threading methods but it comes at a price. Threading an existing serial application with a library-based method is an invasive process requiring significant code modifications. Concurrent tasks must be encapsulated in functions that can be mapped to threads. POSIX and Win32 threads only accept one argument, so it is often necessary to modify function prototypes and data structures.

OpenMP, a compiler-based threading method, provides a high-level interface to the underlying thread libraries. With OpenMP, the programmer uses pragmas (or directives in the case of Fortran) to *describe* parallelism to the compiler. This removes much of the complexity of explicit threading because the compiler handles the details. OpenMP is less invasive too. Significant source code modifications are not usually necessary. A non-OpenMP compiler simply ignores the pragmas, leaving the underlying serial code intact. Much of the fine control over threads is lost, however. Among other things, OpenMP does not give the programmer a way to set thread priorities or to perform event-based or inter-process synchronization. Moreover, OpenMP is a fork-join threading model with an

explicit master-worker relationship among threads. This narrows the range of problems for which OpenMP is suited.

A typical word processor has many opportunities for concurrency. While the user is typing, several background tasks occur simultaneously without interrupting keyboard input. For example, the application periodically saves changes, checks spelling and grammar, and prints documents. This is a good example of functional decomposition, in which different tasks are mapped to threads for concurrent execution. The number of tasks determines the degree of concurrency. The generality and fine control of library-based methods makes them better suited to expressing this type of concurrency. For example, the thread handling keyboard input would be given higher priority than threads handling other, less critical tasks like printing.

OpenMP is designed to express data parallelism, in which threads perform the same task on different data. A web server is a good example of a data parallel application. The same task (servicing HTTP requests) is performed repeatedly on different data (web pages). In a data parallel problem, the amount of data determines the degree of parallelism. The spell checker in a word processor is a good example. The words of the document can be divided among threads, with each thread performing its comparisons independently. The amount of parallel work increases with the number of words in the document.

Advice

In general, OpenMP is best suited to expressing data parallelism while explicit threading methods (i.e., the Pthreads library and the Win32 threading API) are best suited to functional decomposition. Do not try to shoehorn explicit threading methods into a data parallel problem or vice versa, as the following examples illustrate. The following program calculates by numerical integration. The parallelism can be expressed with a single OpenMP pragma. (As mentioned previously, a non-OpenMP compiler will simply ignore the pragma, leaving the underlying serial code intact.)

```
#include <stdio.h>
#define INTERVALS 100000

int main ()
{
   int i;
   float h, x, pi = 0.0;
   h = 1.0 / INTERVALS;

#pragma omp parallel for private(x) reduction(+:pi)
   for (i = 0; i < INTERVALS; i++)
   {
      x = h * (float(i) – 0.5);
      pi += 4.0 / (1.0 + x * x);
   }
   pi *= h;
   printf (“Pi = %f\n”, pi);
}
```

It is possible to express data parallelism with explicit threading methods like Pthreads or the Win32 threading API, but it is not convenient:

```
#include <stdio.h>
#include <pthreads.h>

#define INTERVALS 100000
#define THREADS 4

float global_sum = 0.0;
pthread_mutex_t global_lock = PTHREAD_MUTEX_INITIALIZER;
void *pi_calc (void *num);

int main ()
{
   pthread_t tid[THREADS];
   int i, t_num[THREADS];

   for (i = 0; i < THREADS; i++)
   {
      t_num[i] = i;
      pthread_create (&tid[i], NULL, pi_calc, &t_num[i]);
   }

   for (i = 0; i < THREADS; i++)
      pthread_join (tid[i], NULL);

   printf (“Pi = %f\n”, global_sum);
}

void *pi_calc (void *num)
{
   int i, myid, start, end;
   float h, x, my_sum = 0.0;

   myid = *(int *)num;
   h = 1.0 / INTERVALS;
   start = (INTERVALS / THREADS) * myid;
   end = start + (INTERVALS / THREADS);

   for (i = start; i < end; i++)
   {
      x = h * ((float)i - 0.5);
      my_sum += 4.0 / (1.0 + x * x);
   }
   pthread_mutex_lock (&global_lock);
      global_sum += my_sum;
   pthread_mutex_unlock (&global_lock);
}
```

The size and complexity of the program is increased significantly, and the original serial code is barely recognizable. Notice how the computation must be encapsulated in a function so that it can be mapped to threads. Within this function, the work must be manually divided among the threads.

Explicit threading methods are designed to express functional decomposition, where work is divided by task rather than data. With explicit threading methods, the programmer manually maps concurrent tasks to threads. Consider the standard producer-

consumer problem described in most concurrent programming textbooks. Coding a producer-consumer is straightforward with explicit threading API's because the programmer can dynamically create and destroy threads. Further, synchronization is not limited to just data access. Threads can be made to wait for events. The lack of event-based synchronization makes even this simple problem difficult to code efficiently in OpenMP. The OpenMP `sections` pragma provides some ability to code functional decomposition but the inherent fork-join threading model limits flexibility and scalability. Specifically, the number of parallel sections is fixed at compile-time so the number of producer and/or consumer threads cannot change dynamically at runtime as processor resources change. OpenMP also lacks the ability to assign priorities to threads.

Usage Guidelines

Portability should also be considered when choosing between OpenMP, Pthreads, or Win32 threads. OpenMP-compliant compilers are available for most operating systems, including Windows and Linux. Thread libraries, on the other hand, are not portable. Obviously, the Win32 API is only available on Microsoft operating systems. Even then there are slight differences in supported features between different versions of Windows. The same can be said of Pthreads on Linux and various other flavors of Unix*.

Scalability should be taken into account when threading an application for parallel performance. Does parallelism increase with the number of independent tasks, the amount of data to be processed, or both? Consider an application with only two compute-intensive, independent tasks. On a multiprocessor system with four CPUs, for example, mapping the tasks to Win32 or POSIX threads will only use half of the system. If the two tasks are data parallel, adding OpenMP to each task might be a better solution. If one task is data parallel and the other is not, however, an OpenMP-only solution will not give full system utilization (see any good textbook on parallel computing for a description of Amdahl's Law). A good solution for this example might be to map both independent tasks to a Win32 or POSIX thread then to use OpenMP to express data parallelism within each task.

References

In this chapter, see also:

3.2: Granularity And Parallel Performance

3.3: Load Balance And Parallel Performance

3.6: Use Workload Heuristics To Determine Appropriate Number Of Threads At Runtime

3.8: Exploiting Data Parallelism In Ordered Data Streams

See also:

OpenMP C and C++ Application Program Interface (version 2.0), OpenMP Architecture Review Board, March 2002.

OpenMP Fortran Application Program Interface (version 2.0), OpenMP Architecture Review Board, November 2000.

"Multithreading: Taking Advantage of Intel Architecture-based Multiprocessor Workstations," Intel White Paper, 1999.

"Performance improvements on Intel architecture-based multiprocessor workstations: Multithreaded applications using OpenMP," Intel White Paper, 2000.

"Threading Methodology: Principles and Practices," Intel Technical Report, 2002.

M. Ben-Ari, *Principles of Concurrent Programming*, Prentice-Hall International, 1982.

David R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.

Johnson M. Hart, *Win32 System Programming (2nd Edition)*, Addison-Wesley, 2001.

Jim Beveridge and Robert Wiener, *Multithreading Applications in Win32*, Addison-Wesley, 1997.

3.2 Granularity And Parallel Performance

Category

Application Threading

Scope

General multithreading and performance

Keywords

Granularity, load balance, parallel overhead, VTune, Thread Profiler

Abstract

A key to attaining good parallel performance is choosing the right granularity for your application. Granularity is the amount of work in the parallel task. If granularity is too fine, then performance can suffer from communication overhead. If granularity is too coarse, then performance can suffer from load imbalance. The goal is to determine the right granularity (coarser granularity is usually better) for the parallel tasks, while avoiding load imbalance and communication overhead to achieve the best performance.

Background

The amount of work per parallel task, or granularity, of a multithreaded application greatly affects its parallel performance. When threading an application, the first step is to *partition* the problem into as many parallel tasks as possible. The next step is to determine the necessary *communication* in terms of data and synchronization. In the third step, the performance of the algorithm is considered. Since communication and partitioning are not free operations, one often needs to *agglomerate*, or combine partitions, to overcome the overheads and achieve the most efficient implementation. The agglomeration step is the process of determining the best granularity for the application.

The granularity is often related to how balanced the workload is between threads. It is easier to balance the workload of a large number of small tasks but too many small tasks can lead to excessive parallel overhead. Therefore, coarse granularity is usually best. Increasing granularity too much can create load imbalance, however (see 3.3: Load Balance And Parallel Performance). Tools such as the Intel® Thread Profiler (see 2.5: Using Thread Profiler To Evaluate OpenMP Performance) can help identify the right granularity for your application.

The following examples will show how to improve the performance of a parallel program by decreasing the synchronization overhead and finding the right granularity for the threads. The example used throughout this topic is that of prime number generation (i.e., find all prime numbers between 0 and 1 million). Example Code 1 shows a parallel version using OpenMP.

```
#pragma omp parallel for \
                  schedule(dynamic, 1) \
                  private(j, limit, prime)
for (i = start; i <= end; i += 2)   // Between 0 and 1 million
{
   limit = (int) sqrt((float)i) + 1;
   prime = 1;   // Assume number is prime
   j = 3;
   while (prime && (j <= limit))
   {
      if (i%j == 0) prime = 0;
      j += 2;
   }

   if (prime)
   {
      #pragma omp critical
      {
         number_of_primes++;
         if (i%4 == 1) number_of_41primes++;   // 4n+1 primes
         if (i%4 == 3) number_of_43primes++;   // 4n-1 primes
      }
   }
}
```

Example Code 1. Prime number generation parallelized with OpenMP

This code has both high communication overhead, in the form of synchronization, and a workload that is too small to merit threads. First you will notice a critical section inside the loop to provide a safe mechanism for incrementing the counting variables. The critical section adds synchronization and lock overhead to the parallel loop as shown by the Intel Thread Profiler display in Figure 6a.

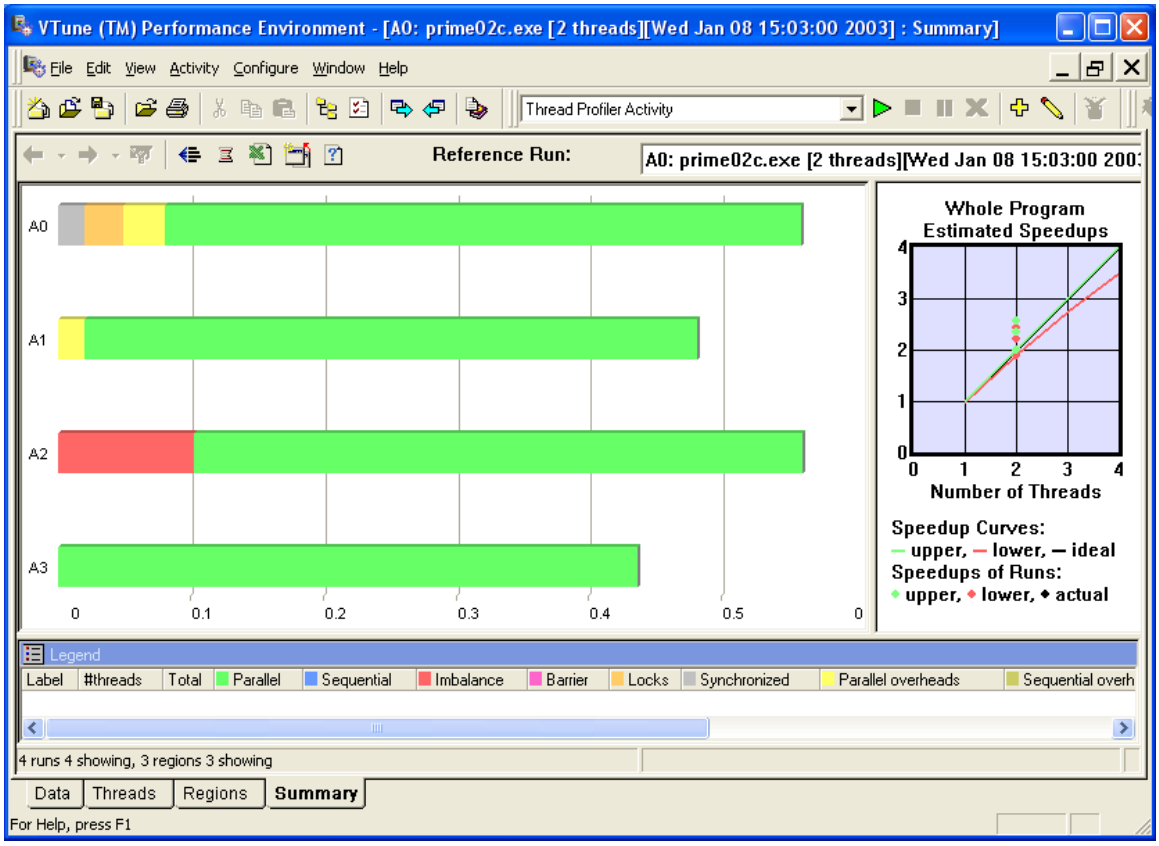

Figure 6. VTune Analyzer Thread Profiler Display

a) A0: 1st run, synchronization and lock overhead, b) A1: 2nd run, parallel overhead, c) A2: 3rd run, load imbalance, d) A3: 4th run, performance problems solved

The lock and synchronization overhead can be removed by replacing the critical section with an OpenMP `reduction` (see Example Code 2). Incrementing counter variables is a common operation, commonly known as a reduction. The OpenMP `reduction` clause provides an efficient way to handle reduction operations.

```
#pragma omp parallel for \
                  schedule(dynamic, 1) private(j, limit, prime) \
                  reduction(+: number_of_primes, \
                               number_of_41primes, \
                               number_of_43primes)
for (i = start; i <= end; i += 2)   // Between 0 and 1 million
{
   limit = (int) sqrt((float)i) + 1;
   prime = 1;   // Assume number is prime
   j = 3;
   while (prime && (j <= limit))
   {

      if (i%j == 0) prime = 0;
      j += 2;
   }

   if (prime)
   {
      number_of_primes++;
      if (i%4 == 1) number_of_41primes++;   // 4n+1 primes
      if (i%4 == 3) number_of_43primes++;   // 4n-1 primes
   }
}
```

Example Code 2. Prime number generation parallelized with OpenMP using the reduction clause instead of critical pragma

The Intel Thread Profiler shows that the lock and synchronization overheads have been eliminated but the parallel overhead is still present (Figure 6b). Dynamic scheduling incurs a small amount of overhead. The `schedule(dynamic, 1)` clause directs the scheduler to dynamically distribute one iteration (i.e., the chunk size) at a time to each thread. Each thread processes a loop iteration then returns to the scheduler to get another iteration. Increasing the chunk size in the `schedule` clause reduces the number of times a thread must return to the scheduler.

If the chunk size is too large, however, load imbalance can occur. For example, the Intel Thread Profiler shows a load imbalance when the chunk size is increased to 100,000 (Figure 6c). Load imbalance occurs because iterations 900,000 to 1,000,000 contain more work than previous chunks. Setting the chunk size to 100 eliminates the parallel overhead and the load imbalance (Figure 6d).

Advice

The parallel performance of a multithreaded application depends on granularity, or the amount of work per parallel task. In general, try to achieve the coarsest granularity possible without creating a load imbalance between threads. Make sure that the amount of work per thread is much larger than the threading overhead. Use the Intel Thread Profiler to find excessive parallel overhead, excessive synchronization, and load imbalance.

Usage Guidelines

While the discussion above makes frequent reference to OpenMP, all of the advice and principles described apply to other threading methods, such as Win32 and POSIX threads.

References

In this series, see also:

Intel® Software Development Tools, 2.5: Using Thread Profiler To Evaluate OpenMP Performance

This chapter, 3.1: Choosing An Appropriate Threading Method: OpenMP Versus Explicit Threading

This chapter, 3.3: Load Balance And Parallel Performance

This chapter, 3.6: Use Workload Heuristics To Determine Appropriate Number Of Threads At Runtime

Synchronization, 4.1: Managing Lock Contention, Large And Small Critical Sections

See also:

Rohit Chandra *et al*., *Parallel Programming in OpenMP*, Morgan Kaufman, 2001.

Ian T. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineers,* Addision-Wesley, 1995.

Ding-Kai Chen *et al*., “The Impact of Synchronization and Granularity on Parallel Systems”, *Proceedings of the 17th Annual International Symposium on Computer Architecture* 1990.

3.3 Load Balance And Parallel Performance

Category

Application Threading

Scope

General multithreading

Keywords

Granularity, load balance, thread scheduling, VTune, Thread Profiler

Abstract

Load balancing the application workload among threads is critical to application performance. The key objective for load balancing is to minimize idle time on threads. Sharing the workload equally across all threads with minimal work sharing overheads results in the shortest critical path of execution, and therefore best performance. Achieving perfect load balance is non-trivial, however, and it depends on the parallelism within the application, workload, the number of threads, load balancing policy, and the threading implementation.

Background

An idle processor during computation is a wasted resource and increases the overall execution time of the computation. This idleness can result from many different causes, such as fetching from memory or I/O. While it may not be possible to completely eliminate a processor from being idle at times, there are measures that programmers can apply to reduce idle time (e.g., overlapped I/O, memory prefetching, reordering data access patterns for better cache utilization).

Similarly, idle threads are also wasted resources in multithreaded executions. An unequal amount of work assigned to threads is a condition known as a load imbalance. The greater the imbalance, the more threads will remain idle and the greater the time needed to complete the computation. The more equitable the distribution of computational tasks to available threads, the lower the overall execution time will be.

As an example, consider a set of twelve independent tasks with the following set of times {10, 6, 4, 4, 2, 2, 2, 2, 1, 1, 1, 1}. Assuming four threads are available for computing this set of tasks, a simple method of task assignment would schedule each thread with three total tasks distributed in order. Thus, Thread-1 would be assigned work totaling 20 time units (10+6+4), Thread-2 would require eight time units (4+2+2), Thread-3 would require five time units (2+2+1), while Thread-4 would be able to execute the three tasks assigned in only three time units (1+1+1). Figure 7 illustrates this distribution of work and shows that the overall execution time for these twelve tasks would be 20 time units.

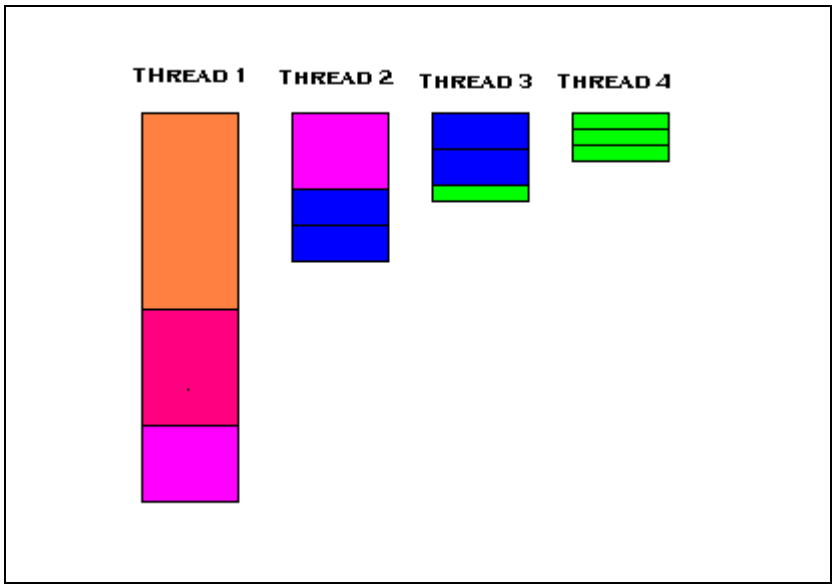

Figure 7. Illustration of task distribution showing load imbalance

A better distribution of work would have been Thread-1 {10}, Thread-2 {6, 1, 1}, Thread-3 {4, 2, 1, 1}, and Thread-4 {4, 2, 2, 2}. This schedule would take only ten time units to complete and would only have two of the four threads idle for two time units each (Figure 8).

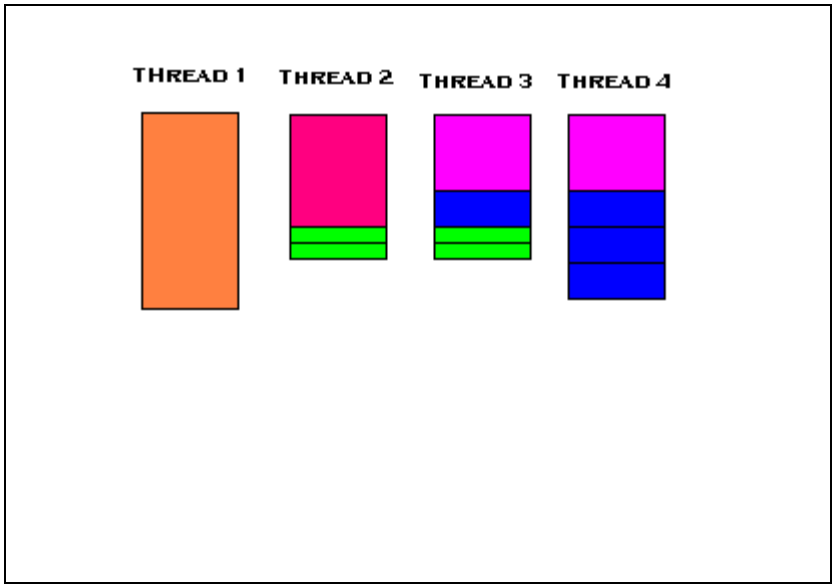

Figure 8. Illustration of task distribution showing better load balance

Advice

For the case when all tasks are the same length, a simple static division of tasks among available threads – dividing the total number of tasks into (nearly) equal-sized groups assigned to each thread – is the best solution. In the general case, though, even when all task lengths are known in advance, finding an optimal, balanced assignment of tasks to

threads is an intractable problem. When the lengths of individual tasks are not the same, dynamic assignment of tasks to threads is a better solution.

OpenMP provides four scheduling methods for iterative work-sharing constructs (see the OpenMP specification for a detailed description of each method). Static scheduling of iterations is used by default. When the amount of work per iteration varies, and the pattern is unpredictable, dynamic scheduling of iterations can better balance the workload. In OpenMP, the dynamic scheduling alternatives are `dynamic` and `guided` which are specified in the `schedule` clause. Under `dynamic` scheduling, chunks of iterations are assigned to threads; when the assignment has been completed, threads request a new chunk of iterations. The optional chunk argument of the `schedule` clause denotes the fixed number of iterations to be assigned under dynamic scheduling. In guided scheduling, iterations are assigned to threads in gradually decreasing chunk sizes. Because of the pattern of assignment, `guided` scheduling requires less overhead than `dynamic` scheduling. The optional chunk argument of the schedule clause denotes the minimum number of iterations to be assigned under guided scheduling.

A special case is when the amount of work per iteration increases (or decreases) monotonically. For example, the number of elements per row in a lower triangular matrix increases regularly. For such cases, setting the chunk size with static scheduling may provide adequate load balance without the added overhead of dynamic or guided scheduling.

When the choice of scheduling method is not apparent, use the `runtime` schedule to specify scheduling method and chunk size at runtime. This allows experimentation without requiring recompilation of the program.

Explicit threading methods (e.g., Win32 and POSIX threads) do not have any means to automatically schedule a set of independent tasks to threads. When needed, such capability must be programmed into the application. Static scheduling of tasks is a straightforward exercise. For dynamic scheduling, two related methods are easily implemented: Producer/Consumer and Manager/Worker. In the former, one or more threads (Producer) places tasks into a queue while the Consumer threads remove tasks to be processed, as needed. While not strictly necessary, the Producer/Consumer model is often used when there is some pre-processing to be done before tasks are made available to Consumer threads. In the Manager/Worker model, Worker threads rendezvous with the Manager thread, whenever more work is needed, to receive assignments directly.

Whatever model is used, consideration must be given to using the correct number and mix of threads to ensure that threads tasked to perform the required computations are not left idle. While a single Manager thread is easy to code and ensures proper distribution of tasks, should Consumer threads stand idle at times, a reduction in the number of Consumers or an additional Producer thread may be needed. The appropriate solution will depend on algorithmic considerations as well as the number and length of tasks to be assigned.

Usage Guidelines

Dynamic scheduling incurs some overhead from parceling out tasks. Bundling small, independent tasks together as a single unit of assignable work can reduce this overhead.

The best choice for how much computation constitutes a task will be based on the computation to be done as well as the number of threads and other resources available at execution time (see 3.2: Granularity And Parallel Performance).

While the discussion above makes frequent reference to OpenMP, all of the advice and principles described apply to other threading methods, such as Win32 and POSIX threads.

References

In this series, see also:

Intel® Software Development Products , 2.5: Using Thread Profiler To Evaluate OpenMP Performance

This chapter, 3.1: Choosing An Appropriate Threading Method: OpenMP Versus Explicit Threading

This chapter, 3.2: Granularity And Parallel Performance

This chapter, 3.6: Use Workload Heuristics To Determine Appropriate Number Of Threads At Runtime

This chapter, 3.9: Manipulate Loop Parameters To Optimize OpenMP Performance

See also:

M. Ben-Ari, *Principles of Concurrent Programming*, Prentice-Hall International, Inc., 1982.

Ian Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

Steven Brawer, *Introduction to Parallel Programming*, Academic Press, Inc., 1989.

3.4 Threading For Turnaround Versus Throughput

Category

Application Threading

Scope

General multithreading

Keywords

Spin-wait, OpenMP, Pthreads, Win32 threads, idle policy

Abstract

Exactly what threads do when waiting for certain kinds of events can make the difference between a fast running application and a slow one, but attention must also be paid to other jobs on the system. Otherwise, the result can be a slow, sluggish system instead of a fast, responsive one. Understanding the usage model of an application can let one optimize for turnaround time of the application or focus on keeping overall system throughput reasonable.

Background

Usage of computers can be classified into two broad categories – dedicated compute engines whose purpose is to produce results as quickly as possible for a single job performing a computation, and dedicated throughput engines whose purpose is to make reasonable progress on all the running jobs. For example, computers performing weather forecasts tend to be dedicated compute engines whereas computers running web servers tend to be throughput engines. Interactive workstations tend to fall somewhere in the middle: for “background” applications the behavior is like throughput engines; for “foreground” applications the behavior is like dedicated compute engines. When designing multithreaded applications, it is very important to understand whether users will run the application expecting high turnaround, high throughput, or perhaps both. Once the usage is understood, the application can be designed to favor a particular scenario, switch between the scenarios, or function reasonably well in both.

Threads in multithreaded programs communicate by exchanging data through shared resources. Pthreads provides condition variables, semaphores, and mutexes for this purpose, whereas the Win32 threading API provides events, semaphores, mutexes and a specialized form of mutex variable called a critical section. The programmer can also create such resources using a memory location as a flag to communicate between cooperating threads, and carefully writing to the location using some kind of volatile or acquire/release semantics. Regardless of the underlying method, when a thread tries to acquire such a resource and a different thread already holds that resource (in an exclusive state), the acquiring thread must wait. What the thread does when waiting is crucial to the performance of the application and the overall system. The two extreme cases of what threads do when waiting are: *spin-waiting*, in which the thread keeps the processor busy and repeatedly checks on the resource to see if it has become free; *blocking*, in which the thread immediately relinquishes the CPU to the operating system and asks to be woken up when the resource becomes available. Modern implementations provide a middle

ground between these extremes with adaptive switching from spin-waiting to yielding to blocking so other jobs can progress.

Different functions perform different kinds of operations when waiting. For example, older Linux Pthreads wait functions spin-wait when waiting, whereas the Win32 `WaitForSingleObject` and `WaitForMultipleObjects` functions block and the Win32 `EnterCriticalSection` function spin waits for a user-controllable period then blocks on an associated kernel object. The OpenMP API facilitates synchronous, compute-bound applications that usually allocate no more than one thread per processor. Thus, the OpenMP `critical` and `ordered` constructs and the lock API typically spin-wait. The Intel OpenMP implementation provides controls to adjust the period for which a thread spin-waits before blocking.

Advice

If resources are held for very short periods of time (e.g., a few hundred clock cycles), it is usually better to employ a spin-wait because the overhead of relinquishing the CPU to the operating system may be greater than the time that the resource is held (see Synchronization, 4.1: Managing Lock Contention, Large And Small Critical Sections). The Windows critical section functions (see Synchronization, 4.3: Win32 Atomics Versus User-Space Locks Versus Kernel Objects For Synchronization) and the OpenMP critical constructs and locks API serve this purpose well.

For compute-bound applications running on dedicated systems, where the number of simultaneous active threads is expected to be less than or equal to the number of processors, using an API that spin-waits for at least a short period of time will usually result in a better performing application. Spin waiting is usually non-disruptive on an otherwise idle CPU. However, performing a spin-wait on a virtual processor on a Hyper-Threading Technology-enabled CPU can be disruptive to the other virtual processors on the CPU. The OpenMP implementation in the Intel compilers is ideal for such types of applications. The OpenMP runtime library automatically adjusts the spin parameters to account for Hyper-Threading Technology. The Windows critical section functions with the user controlled spin counts can also serve these applications (see Synchronization, 4.1: Managing Lock Contention, Large And Small Critical Sections).

Conversely, for throughput-oriented applications, or for applications where the number of active threads exceeds the number of processors on the system, a blocking API will result in better overall throughput because blocking ensures that other ready-to-run threads in the application or other jobs on the system can run immediately. The Windows semaphore, event, and mutex variables provide appropriate functionality for this class of application.

Most modern locking algorithms do not spin-wait indefinitely. They usually employ a “back-off” scheme whereby after spinning for some time, they relinquish the CPU to the operating system. Except for specialized situations like real-time applications, when designing your own locks via memory operations, you should design an appropriate “back-off” scheme to avoid bogging down the entire system, which could happen with pure spin-waits.

Another important point to keep in mind when designing your own spin-waiting loops is the use of the PAUSE instruction inside the spin-wait loop on Pentium® 4 processors. The PAUSE instruction is a low latency instruction that releases the processor bus for use by other processors in a multi-processor configuration. On CPU's with Hyper-Threading Technology, PAUSE makes spin-waiting less disruptive to the other virtual processors on the CPU. On systems where spin-waiting is non-disruptive in the processor, PAUSE has no effect.

For OpenMP applications, use the Intel® compilers and set the environment variable KMP_LIBRARY=turnaround to spin-wait with a back-off algorithm and use KMP_LIBRARY=throughput to spin-wait with back-off algorithm that eventually yields the CPU to the operating system.

Usage Guidelines

Spin-waiting consumes CPU cycles. Nevertheless, it can be a good technique for reducing turnaround time when you expect to acquire the resource that you are waiting on quickly. This is true because acquiring a lock is much faster than getting woken up by another thread via events or condition variables. When long waits are expected, spin-waiting can disrupt other jobs and result in sluggish system-wide performance. When spin-waiting is used, it should only be used for short periods of time (typically on the order of hundredths of a second) to avoid such problems. On CPUs with Hyper-Threading Technology, spin-waiting can be especially wasteful because the virtual processors share execution resources. On such systems, it is very important to minimize the disruption of the virtual processors by using PAUSE instruction in spin-wait loops and by tuning spin-wait counts to very low values. The OpenMP runtime library in the Intel compilers makes these adjustments automatically.

References

In this manual, see also:

> Synchronization, 4.1: Managing Lock Contention, Large And Small Critical Sections
>
> Synchronization, 4.2: Use Synchronization Routines Provided By The Threading API Rather Than Hand-Coded Synchronization
>
> Synchronization, 4.3: Win32 Atomics Versus User-Space Locks Versus Kernel Objects For Synchronization

3.5 Expose Parallelism By Avoiding Or Removing Artificial Dependencies

Category

Application Threading

Scope

General multithreading but especially data decomposition and OpenMP

Keywords

Data dependencies, compiler optimizations, blocking algorithms, Win32 Threads, OpenMP, Pthreads

Abstract

Many applications and algorithms contain serial optimizations that inadvertently introduce data dependencies and inhibit parallelism. One can often remove such dependences, through simple transforms, or even avoid them altogether, through techniques such as domain decomposition or blocking.

Background

While multithreading for parallelism is an important source of performance, it is equally important to ensure that each thread runs efficiently. While optimizing compilers do the bulk of this work, it is not uncommon for programmers to make source code changes that improve performance by exploiting data reuse and selecting instructions that favor machine strengths. Unfortunately, the same techniques that improve serial performance can inadvertently introduce data dependencies that make it difficult to achieve additional performance through multithreading.

One example is the reuse of intermediate results to avoid duplicate computations. As an example, softening an image through blurring can be achieved by replacing each image pixel by a weighted average of the pixels in its neighborhood, itself included. Example Code 3 below shows pseudo-code describing a 3 x 3 blurring stencil.

```
for each pixel in (imageIn)
   sum = value of pixel
   // compute the average of 9 pixels from imageIn
   for each neighbor of (pixel)
      sum += value of neighbor
   // store the resulting value in imageOut
   pixelOut = sum / 9
```

Example Code 3. Pseudo-code describing a 3 x 3 blurring stencil

The fact that each pixel value feeds into multiple calculations allows one to exploit data reuse for performance. In the following pseudo-code, intermediate results are computed and used three times, resulting in better serial performance:

```
subroutine BlurLine(lineIn, lineOut)
   for each pixel j in (lineIn)
      // compute the average of 3 pixels from line
      // and store the resulting value in lineout
      pixelOut = (pixel j-1 + pixel j + pixel j+1) / 3

declare lineCache[3]
lineCache[0] = 0
BlurLine(line 1 of imageIn, lineCache[1])
for each line i in (imageIn)
   BlurLine (line i+1 of imageIn, lineCache[i mod 3])
   lineSums = lineCache[0] + lineCache[1] + lineCache[2]
   lineOut = lineSums / 3
```

This optimization introduces a dependence between the computations of neighboring lines of the output image. If one attempts to compute the iterations of this loop in parallel, the dependencies will cause incorrect results.

Another common example is pointer offsets inside a loop (). By incrementing `ptr`, the code potentially exploits the fast operation of a register increment and avoids the arithmetic of computing `someArray[i]` for each iteration. While each call to compute may be independent of the others, the pointer becomes an explicit dependence – its value in each iteration depends on that in the last. If this loop is parallelized with OpenMP, for example, the Intel Thread Checker will report memory conflicts on the use of `ptr`.

```
ptr = &someArray[0];

for (i = 0; i < N; i++)
{
   Compute (ptr);
   ptr++;
}
```

Example Code 4. Pointer offsets inside a loop

Finally, there are often situations where the algorithms invite parallelism, but the data structures have been designed to a different purpose that unintentionally prevents parallelism. Sparse matrix algorithms are one such example. Because most matrix elements are zero, the usual matrix representation is often replaced with a “packed” form, consisting of element values and relative offsets, used to skip zero-valued entries.

This article aims to present strategies to effectively introduce parallelism in these challenging situations.

Advice

Naturally, it's best to find ways to exploit parallelism without having to remove existing optimizations or make extensive source code changes. Before removing any serial optimization to expose parallelism, consider whether applying the existing kernel to a subset of the overall problem can preserve the optimization. Normally, if the original algorithm contains parallelism, it is also possible to define subsets as independent units and compute them in parallel.

To efficiently thread the blurring operation, consider subdividing the image into sub-images, or blocks, of fixed size. The blurring algorithm allows the blocks of data to be computed independently. The following pseudo-code illustrates the use of image blocking:

```
// One time operation:
// Decompose the image into non-overlapping blocks.
blockList = Decompose (image, xRes, yRes)
foreach (block in blockList)
{
   BlurBlock (block, imageIn, imageOut)
}
```

The existing code to blur the entire image can be reused in the implementation of `BlurBlock`. Using OpenMP or explicit threads to operate on multiple blocks in parallel yields the benefits of multithreading and retains the original optimized kernel.

In other cases, the benefit of the existing serial optimization is small compared to the overall cost of each iteration, making blocking unnecessary. This is often the case when the iterations are sufficiently coarse-grained to expect a speedup from parallelization. The pointer increment example is one such instance. The induction variables can be easily replaced with explicit indexing, removing the dependence and allowing simple parallelization of the loop.

```
ptr = &someArray[0];

for (i = 0; i < N; i++)
{
   Compute (ptr[i]);
}
```

Note that the original optimization, though small, is not necessarily lost. Compilers often optimize index calculations aggressively – by utilizing increment or other fast operations – allowing you to enjoy the benefits of both serial and parallel performance.

Other situations, such as code involving packed sparse matrices can be more challenging to thread. Normally, it is not practical to unpack data structures but it is often possible to subdivide the matrices into blocks, storing pointers to the beginning of each block. When these matrices are paired with appropriate block-based algorithms, the benefits of a packed representation and parallelism can be simultaneously realized.

The blocking techniques described above are a case of a more general technique called domain decomposition. After decomposition, each thread works independently on one or more domains. In some situations, the nature of the algorithms and data dictate that the work per domain will be nearly constant. In other situations, the amount of work may vary from domain to domain. In these cases, if the number of domains equals the number of threads, parallel performance can be limited by load imbalance. In general, it is best to ensure that the number of domains is reasonably large compared to the number of threads. This will allow techniques such as dynamic scheduling to balance the load across threads.

Usage Guidelines

Some serial optimizations deliver large performance gains. Consider the number of processors you are targeting to ensure that speedups from parallelism will outweigh the performance loss of any lost optimization.

Introducing blocking algorithms can sometimes hinder the compiler's ability to distinguish aliased from unaliased data. If, after blocking, the compiler can no longer determine that data is unaliased, performance may suffer. Consider using the `restrict` keyword to explicitly prohibit aliasing (see Intel® Software Development Products, 2.1: Automatic Parallelization With Intel® Compilers). Enabling inter-procedural optimizations also helps the compiler detect unaliased data.

References

In this series, see also:

> Intel® Software Development Products, 2.1: Automatic Parallelization With Intel® Compilers
>
> Intel® Software Development Products, 2.4: Find Multithreading Errors With The Intel Thread Checker
>
> This chapter, 3.2: Granularity And Parallel Performance
>
> This chapter, 3.3: Load Balance And Parallel Performance

3.6 Use Workload Heuristics To Determine Appropriate Number Of Threads At Runtime

Category

Application Threading

Scope

General multithreading, OpenMP, POSIX threads, Win32 threads

Keywords

Load balance, granularity, Win32 Threads, OpenMP, Pthreads

Abstract

Most application and workload pairs have a finite amount of work, and therefore a finite speedup due to multithreading. Choosing the right number of threads can be an important consideration in the performance delivered by multithreaded applications. This article will discuss the factors involved in designing a heuristic to choose an appropriate number of threads.

Background

When applications are threaded for functionality, programmers often dedicate particular functions to particular threads, and it is rare for all of the threads to be active at the same time. In such functionally threaded systems, the choice of the number of threads is often based on the functionality desired, and is not easily varied. Fortunately, this choice is normally not performance-critical.

For applications – or portions of applications – that have been threaded for performance reasons, however, programmers often have the ability to choose how many threads to apply to a problem. Most applications cannot use an arbitrary number of threads effectively based on various implicit and explicit costs associated with threading. For example, the implicit costs include the extra scheduling burden on the operating system, the cost of migrating data to the thread and the increased memory pressure on the system to keep all the threads supplied with data. Explicit costs include thread startup, shutdown, and coordination. These costs, together with the amount of work, the number of independent work items available for parallel execution, and their granularity, play an important part in choosing an appropriate number of threads to apply to a problem.

When using operating system threads, the programmer makes this decision explicitly by creating and using the number of threads desired. When using OpenMP, however, programmers sometimes let the system decide how many threads to use, and most OpenMP implementations (including the Intel implementation) default to the number of processors on the system. For most applications, this is not the best choice because they are unlikely to scale to the entire range of parallel systems available, from single-CPU systems with Hyper-Threading Technology all the way to 64-CPU and larger SMP systems.

For all these reasons, it is best to let either let the user determine the number of threads to use, or to use runtime heuristics or measurements to understand the size of the computation and data and then choose an appropriate number of threads.

Advice

For applications where the workload depends on application input that can vary widely, defer the decisions of how many threads to employ until runtime when the input sizes can be examined. Examples of workload input parameters that affect the thread count include things like matrix size, database size, image/video size and resolution, depth/breadth/bushiness of tree based structures, and size of list based structures.

Similarly, for applications designed to run on systems where the processor count can vary widely, defer the number of threads to employ decision till application run-time when the machine size can be examined.

Using the above workload and system size inputs, heuristics should be developed, based on empirical data, to set the thread count at application run-time.

For applications where the amount of work is unpredictable from the input data, consider using a calibration step to understand the workload and system characteristics to aid in choosing an appropriate number of threads. If the calibration step is expensive, it can be made persistent by storing in a permanent place like the file system.

Avoid creating more threads than the number of processors on the system, when all the threads can be active simultaneously. This situation causes the operating system to multiplex the processors and typically yields sub-optimal performance.

When developing a library as opposed to an entire application, provide a mechanism whereby the user of the library can conveniently select the number of threads used by the library, because it is possible that the user has higher-level parallelism that renders the parallelism in the library unnecessary or even disruptive.

Finally, for OpenMP, use the `num_threads` clause on parallel regions to control the number of threads employed and use the `if` clause on parallel regions to decide whether to employ multiple threads at all. The `omp_set_num_threads` function can also be used but it is not recommended except in specialized well-understood situations because its affect is global and persists even after the current function ends, possibly affecting parents in the call tree. The `num_threads` clause is local in its effect and so does not impact the calling environment.

Usage Guidelines

With each new generation of computer systems, the implicit and explicit costs can change because of underlying changes in CPU to memory speed ratios, different algorithms, and the topological layout of systems, from simple SMP systems to multithreaded SMP systems to NUMA systems and combinations of each. Such changes can require a reevaluation of the number of threads to use. This can be a particularly vexing problem for applications with fine-grained parallelism, because these tend to be particularly sensitive to the issues listed. Applications with coarse-grained parallelism tend to be more stable in this regard and ought to be favored.

In addition to the application specific factors considered here, it is important to pay attention to the computing environment. For systems dedicated to running just a single application, the heuristic for the number of threads chosen can be quite different than for systems shared with other jobs.

References

In this series, see also:

Intel Software Development Products, 2.4: Using Thread Profiler To Evaluate OpenMP Performance

This chapter, 3.2: Granularity And Parallel Performance

This chapter, 3.3: Load Balance And Parallel Performance

This chapter, 3.4: Threading For Turnaround Versus Throughput

3.7 Reduce System Overhead With Thread Pools

Category

Application Threading

Scope

General multithreading

Keywords

Thread pool, system overhead, Win32 threads, OpenMP, Pthreads

Abstract

Many threaded applications manage their threads with a threads-on-demand policy. With this policy, threads are created as needed and deleted immediately after use. A key benefit of the policy is the simplicity of coding and thread management. However, creating many threads during execution can complicate the control logic of a program in order to account for instances when the operating system is unable to create a thread. Many applications ignore the possibility of such failures and are potentially unsafe. Further, frequent thread creation causes performance penalties, as the cost of creating a thread is substantial. The thread management cost can be very high for applications dealing with many threads, including, for example, server applications. As the number of threads increases, thread creation, termination, scheduling, and context-switching costs can increase to the point where benefits of multithreading are overcome by system overhead.

Background

Thread pools offer a cost-effective approach to managing threads. A thread pool is a group of threads waiting for work assignments. In this approach, threads are created once during an initialization step and terminated during a finalization step. This simplifies the control logic for checking for failures in thread creation midway through the application and amortizes the cost of thread creation over the entire application. Once created, the threads in the thread pool wait for work to become available. Other threads in the application assign tasks to the thread pool. Typically, this is a single thread called the thread manager or dispatcher. After completing the task, each thread returns to the thread pool to await further work. Depending upon the work assignment and thread pooling policies employed, it is possible to add new threads to the thread pool if the amount of work grows. This approach has obvious benefits:

- Possible runtime failures midway through application execution due to inability to create threads can be avoided with simple control logic.
- Thread management costs from thread creation are minimized. This in turn leads to better response times for processing workloads and allows for multithreading of finer-grained workloads (see this chapter, 3.2: Granularity And Parallel Performance).

A typical usage scenario for thread pools is in server applications, which often launch a thread for every new request. A better strategy is to queue service requests for processing by an existing thread pool. A thread from the pool grabs a service request from the queue, processes it, and returns to the queue to get more work.

Thread pools can also be used to perform overlapping asynchronous I/O. The I/O completion ports provided with the Win32 API allow a pool of threads to wait on an I/O completion port, and process packets from overlapped I/O operations.

OpenMP* is strictly a fork/join threading model. In some OpenMP implementations, threads are created at the start of a parallel region and destroyed at the end of the parallel region. OpenMP applications typically have several parallel regions with intervening serial regions. Creating and destroying threads for each parallel region can result in significant system overhead, especially if a parallel region is inside a loop. Therefore, the Intel OpenMP implementation uses thread pools. A pool of worker threads is created at the first parallel region. These threads exist for the duration of program execution. More threads may be added automatically if requested by the program. The threads are not destroyed until the last parallel region is executed.

Thread pools can be created on Windows and Linux using the thread creation API. For instance, a custom thread pool using Win32 threads may be created as follows:

```
// Initialization method/function
{
   DWORD tid;
   //
   // Create initial pool of threads
   //
   for (int i = 0; i < MIN_THREADS; i++)
   {
      HANDLE *ThHandle = CreateThread (NULL,
                                       0,
                                       CheckPoolQueue,
                                       NULL,
                                       0,
                                       &tid);
      if (ThHandle == NULL)
          // Handle Error
      else
          RegisterPoolThread (ThHandle);
   }
}
```

The function `CheckPoolQueue` executed by each thread in the pool is designed to enter a wait state until work is available on the queue. The thread manager can keep track of pending jobs in the queue and dynamically increase the number of threads in the pool based on the demand.

Advice

Use thread pools to minimize thread-management overheads, and to improve application performance (i.e., throughput, response time, and scalability).

The Intel OpenMP implementation already uses thread pools to minimize overhead. OpenMP is well suited for synchronous threaded applications, particularly data parallel applications (see this chapter, 3.1: Choosing An Appropriate Threading Method: OpenMP Versus Explicit Threading).

Use thread pools with I/O completion ports to improve asynchronous I/O performance in Windows applications.

Applications can create and manage thread pools using the Win32 and POSIX threads API's. Standard thread pool functions/classes are available with Win32, C# in .Net, Java, and RogueWave.

References

In this document, see also:

This chapter, 3.1: Choosing An Appropriate Threading Method: OpenMP Versus Explicit Threading

This chapter, 3.2: Granularity And Parallel Performance

This chapter, 3.4: Threading For Turnaround Versus Throughput

See also:

Win32 API: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/queueuserworkitem.asp

C# with .Net: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemThreadingThreadPoolClassTopic.asp

RogueWave: http://www.roguewave.com/support/docs/sourcepro/threadsug/3-7.html

3.8 Exploiting Data Parallelism In Ordered Data Streams

Category

Application Threading

Scope

General multithreading, any programming language or operating system

Keywords

Data parallelism, I/O, order dependence

Abstract

Many compute-intensive applications involve complex transformations of ordered input data to ordered output data. Examples include sound and video transcoding, lossless data compression, and even seismic data processing. While the algorithms employed in these transformations are often parallel, managing the I/O order dependence can be a challenge. This section identifies some of these challenges and illustrates strategies for addressing them, all while maintaining parallel performance.

Background

Consider the problem of threading a video compression engine designed to perform real-time processing of uncompressed video from a live video source to disk or a network client. Clearly, harnessing the power of multiple processors can be key to meeting the real-time requirements of such an application.

Video compression standards such as MPEG-2 and MPEG-4 are designed for streaming over unreliable links. Consequently, it is easy to treat a single video stream as a sequence of smaller, standalone streams. One can achieve substantial speedups by processing these smaller streams in parallel. Some of the challenges in exploiting this parallelism through multithreading include:

1. Defining non-overlapping subsets of the problem and assigning them to threads
2. Ensuring the input data is read exactly once and in the correct order
3. Outputting blocks in the correct order, regardless of the order in which processing actually completes and without significant performance penalties
4. Performing the above without *a priori* knowledge of the actual extent of the input data.

In other situations, such as lossless data compression, it is often possible to determine the input data size in advance and to explicitly partition the data into independent input blocks. The techniques outlined here apply equally well to this case.

Advice

The temptation might be to set up a chain of producers and consumers, but this approach is not scalable and is vulnerable to load imbalance. Instead, we will address each of the challenges above to achieve a more scalable design using data decomposition.

The basic approach is to create a team of threads, with each thread reading a block of video, encoding it, and outputting it to a reorder buffer. Upon completion of each block, a

thread returns to read and process the next block of video, and so on. This dynamic allocation of work minimizes load imbalance. The reorder buffer ensures that blocks of coded video are written in the correct order, regardless of their order of completion.

The original video encoding algorithm might take this form:

```
inFile = OpenFile ()
outFile == InitializeOutputFile ()
WriteHeader (outFile)
outputBuffer = AllocateBuffer (bufferSize)

while (frame = ReadNextFrame (inFile))
{
   EncodeFrame (frame, outputBuffer)
   if (outputBuffer size > bufferThreshold)
      FlushBuffer(outputBuffer, outFile)
}
FlushBuffer (outputBuffer, outFile)
```

The first task is to replace the read and encode frame sequence with a block-based algorithm. This sets up the problem for decomposition across a team of threads:

```
WriteHeader (outFile)

while (block = ReadNextBlock (inFile))
{
   while(frame = ReadNextFrame (block))
   {
      EncodeFrame (frame, outputBuffer)
      if (outputBuffer size > bufferThreshold)
         FlushBuffer (outputBuffer, outFile)
   }
   FlushBuffer (outputBuffer, outFile)
}
```

The definition of a block of data will vary from one application to another, but in the case of a video stream, a natural block boundary might be the first frame at which a scene change is detected in the input, subject to constraints of minimum and maximum block sizes. Block-based processing requires allocation of an input buffer and minor changes to the source code to fill the buffer before processing. Likewise, the `ReadNextFrame` method must be changed to read from the buffer rather than the file.

The next step is to change the output buffering strategy to ensure that entire blocks are written as a unit. This approach simplifies output reordering substantially, since we need only ensure that the blocks are output in the correct order. The following code reflects the change to block-based output:

```
WriteHeader (outFile)

while (block = ReadNextBlock (inFile))
{
```

```
   while (frame = ReadNextFrame (block))
   {
      EncodeFrame (frame, outputBuffer)
   }
   FlushBuffer (outputBuffer, outFile)
}
```

Depending on the maximum block size, a larger output buffer may be required.

Because each block is independent of the others, a special header typically begins each output block. In the case of an MPEG video stream, this header precedes a complete frame, known as an I-frame, relative to which future frames are defined. Consequently, the header is moved inside the loop over blocks:

```
while (block = ReadNextBlock (inFile))
{
   WriteHeader (outputBuffer)
   while (frame = ReadNextFrame (block))
   {
      EncodeFrame (frame, outputBuffer)
   }
   FlushBuffer (outputBuffer, outFile)
}
```

With these changes, it is possible to introduce parallel threads, using a thread library (i.e., Pthreads or the Win32 threading API) or OpenMP parallel sections[2]:

```
// Create a team of threads with private copies of outputBuffer,
// block, and frame and shared copies of inFile and outFile
while (AcquireLock,
       block = ReadNextBlock (inFile),
       ReleaseLock, block)
{
   WriteHeader (outputBuffer)
   while (frame = ReadNextFrame (block))
   {
      EncodeFrame (frame, outputBuffer)
   }
   FlushBuffer (outputBuffer, outFile)
}
```

This is a simple but effective strategy for reading data safely and in order. Each thread acquires a lock, reads a block of data, then releases the lock. Sharing the input file ensures that blocks of data are read in order and exactly once. Because a ready thread always acquires the lock, the blocks are allocated to threads on a dynamic, or first-come-first-served basis, which typically minimizes load imbalance.

The final task is to ensure that blocks are output safely and in the correct order. A simple strategy would be to use locks and a shared output file to ensure that only one block is written at a time. This approach ensures thread-safety, but would allow the blocks to be

[2] The code can be made even simpler using the Intel WorkQueue extensions to OpenMP.

output in something other than the original order. Alternately, threads could wait until all previous blocks have been written before flushing their output. Unfortunately, this approach introduces inefficiency because a thread sits idle waiting for its turn to write.

A better approach is to establish a circular reorder buffer for output blocks[3]. Each block is assigned a sequential serial number. The "tail" of the buffer establishes the next block to be written. If a thread finishes processing a block of data other than that pointed to by the tail, it simply enqueues its block in the appropriate buffer position and returns to read and process the next available block. Likewise, if a thread finds that its just-completed block is that pointed to by the tail, it writes that block and any contiguous blocks that were previously enqueued. Finally, it updates the buffer's tail to point to the next block to be output. The reorder buffer allows completed blocks to be enqueued out-of-order, while ensuring they are written in order.

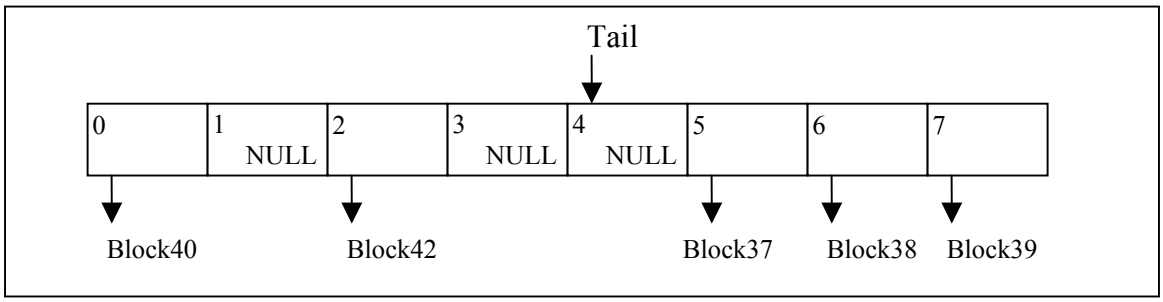

Figure 9. State of example reorder buffer before writing

Figure 9 illustrates one possible state of the reorder buffer. Blocks 0 through 35 have already been processed and written, while blocks 37, 38, 39, 40 and 42 have been processed and are enqueued for writing. When the thread processing block 36 completes, it writes out blocks 36 through 40, leaving the reorder buffer in the state shown in Figure 10. Block 42 remains enqueued until block 41 completes.

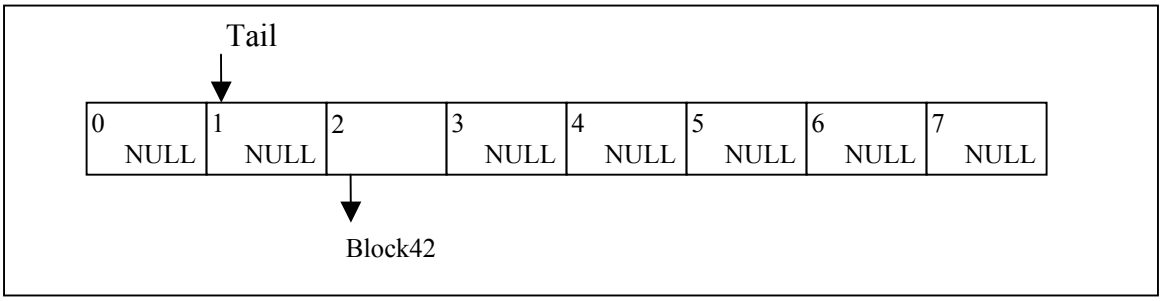

Figure 10. State of example reorder buffer after writing

Naturally, one needs to take certain precautions to ensure the algorithm is robust and fast:

- The shared data structures must be locked when read or written.

[3] This approach is analogous to the reorder buffers used in some microprocessors to allow instructions to be processed out of order but retired in order.

- The number of slots in the buffer must exceed the number of threads.
- Threads must efficiently wait, if an appropriate slot is not available in the buffer.
- Pre-allocate multiple output buffers per thread. This allows one to enqueue a pointer to the buffer and avoids extraneous data copies and memory allocations.

Using the output queue, the final algorithm is:

```
inFile = OpenFile ()
outFile == InitializeOutputFile ()

// Create a team of threads with private
// copies of outputBuffer, block, and frame, shared
// copies of inFile and outFile.
while (AcquireLock,
       block = ReadNextBlock (inFile),
       ReleaseLock, block)
{
   WriteHeader (outputBuffer)
   while (frame = ReadNextFrame (block))
   {
      EncodeFrame (frame, outputBuffer)
   }
   QueueOrFlush (outputBuffer, outFile)
}
```

This algorithm allows in-order I/O but still affords the flexibility of high performance, out-of-order parallel processing.

Usage Guidelines

In some instances, the time to read and write data is comparable to the time required to process the data. In these cases, the following techniques may be beneficial:

Asynchronous I/O – Linux and Windows provide APIs to initiate a read or write and later wait on or be notified of its completion. Using these interfaces to "pre-fetch" input data and "post-write" output data while performing other computation can effectively hide I/O latency. On Windows, files are opened for asynchronous I/O by providing the `FILE_FLAG_OVERLAPPED` attribute. On Linux, asynchronous operations are effected through a number of `aio_*` functions provided by `libaio`.

When the amount of input data is significant, static decomposition techniques can lead to physical disk "thrashing", as the hardware attempts to service a number of concurrent but non-contiguous reads. Following the advice above of a shared file descriptor and a dynamic, first-come-first-served scheduling algorithm can enforce in-order, contiguous reads, which in turn improve overall I/O subsystem throughput.

It is important to carefully choose the size and number of data blocks. Normally, a large number of blocks affords the most scheduling flexibility, which can reduce load imbalance. On the other hand, very small blocks can introduce unnecessary locking overhead and even hinder the effectiveness of data compression algorithms. See the load balance and granularity sections of this document for more advice on choosing the number and size of blocks, relative to the number of threads.

References

In this series, see also:

Intel Software Development Products, 2.4: Find Multithreading Errors With The Intel Thread Checker

This chapter, 3.2: Granularity And Parallel Performance

Synchronization, 4.1: Managing Lock Contention, Large And Small Critical Sections

Synchronization, 4.4: Use Non-Blocking Locks When Possible

3.9 Manipulate Loop Parameters To Optimize OpenMP Performance

Category

Application Threading

Scope

OpenMP applications on any operating system

Keywords

Loop optimizations, granularity, load balance, OpenMP, barrier

Abstract

In data-parallel applications, the same independent operation is performed repeatedly on different data. Loops are usually the compute-intensive segments of data-parallel applications so loop optimizations directly impact performance.

Background

Loop optimizations offer a good opportunity to improve the performance of data-parallel applications. These optimizations, such as loop fusion, loop interchange, and loop unrolling, are usually targeted at improving granularity, load balance, and data locality, while minimizing synchronization and other parallel overhead. As a rule of thumb, loops with high trip counts are the best candidates for parallelization. A higher trip count enables better load balance due to the greater availability of tasks that can be distributed among the threads. The amount of work per loop iteration is also a factor, however. Unless otherwise stated, we shall assume that the amount of work in each iteration is (roughly) equal to every other iteration in the same loop.

Consider the scenario of a loop using the OpenMP `for` work-sharing construct, shown in Figure 11 . In this case, the low trip count leads to a load imbalance (see this chapter, 3.3: Load Balance And Parallel Performance) when the loop iterations are distributed over four threads. If a single iteration takes only a few seconds, this imbalance may not cause a significant impact. However, if each iteration takes an hour, three of the threads remain idle for 60 minutes while the fourth continues working. Contrast this to 1003 one-hour iterations and four threads. In this case, a single hour of idle time after ten days of execution is insignificant.

```
#pragma omp for
  for (i = 0; i < 13; i++)
  {
  // Computation
  }
```

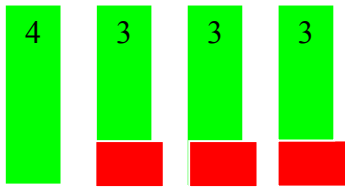

Figure 11. Parallelizing loops with low trip count sometimes lead to load imbalance

Advice

For multiply nested loops, choose the outermost loop that is safe to parallelize. This generally gives the coarsest granularity (see this chapter, 3.2: Granularity And Parallel Performance). Ensure that work can be evenly distributed to each thread. If this is not possible because the outermost loop has a low trip count, an inner loop with a high trip count may be a better candidate for threading, e.g.:

```
void copy (int imx, int jmx, int kmx,
           double**** w, double**** ws)
{
   for (int nv = 0; nv < 5; nv++)
      for (int k = 0; k < kmx; k++)
         for (int j = 0; j < jmx; j++)
            for (int i = 0; i < imx; i++)
               ws[nv][k][j][i] = w[nv][k][j][i];
}
```

With any number of threads besides five, parallelizing the outer loop will result in load imbalance and idle threads. The inefficiency would be especially severe if the array dimensions `imx`, `jmx`, and `kmx` are very large. Parallelizing an inner loop is probably a better option in this case.

Merging nested loops to increase the iteration count is another loop optimization that can help parallel performance. For example, two nested loops with trip counts of 8 and 9, respectively, can be combined into a single loop of 72 iterations (Figure 12). However, if both loop counters are used to index arrays, the new loop counter must be translated back into the corresponding index values. This creates extra operations that original nested loop did not have. This slight increase in work is offset, however, by the loss of overhead from one loop and the greater parallelism that is exposed by merging the two loops into one.

```
#pragma omp parallel for
   for (i = 0; i < 8; i++)
      for (j = 0; j < 9; j++)
         a[i][j] = b[j] *
c[i];
```

```
#pragma omp parallel for
   for (ij = 0; ij < 72;
ij++)
   {
      int i = ij / 9;
      int j = ij % 9;
      a[i][j] = b[j] * c[i];
   }
```

Figure 12. Merging nested loops to increase trip count can expose more parallelism and help performance

Avoid the implicit barrier at the end of OpenMP work-sharing constructs when it is safe to do so. All OpenMP work-sharing constructs (`for`, `sections`, `single`) have an implicit barrier at the end of their structured blocks. All threads must rendezvous at this barrier before execution can proceed. Sometimes these barriers are unnecessary and negatively impact performance. Use the OpenMP `nowait` clause to disable this barrier as shown the following example:

```
void copy (int imx, int jmx, int kmx,
           double**** w, double**** ws)
{
   #pragma omp parallel shared(w, ws)
   {
      for (int nv = 0; nv < 5; nv++)
         for (int k = 0; k < kmx; k++)   // kmx is usually small
            #pragma omp for shared(nv, k) nowait
            for (int j = 0; j < jmx; j++)
               for (int i = 0; i < imx; i++)
                  ws[nv][k][j][i] = w[nv][k][j][i];
   }
}
```

Since the computations in the innermost loop are all independent, there is no reason for threads to wait at the implicit barrier before going on to the next `k` iteration. If the amount of work per iteration is unequal, the `nowait` clause allows threads to proceed with useful work rather than sit idle at the implicit barrier.

Use the OpenMP `if` clause to choose serial or parallel execution based on runtime information (see this chapter, 3.6: Use Workload Heuristics To Determine Appropriate Number Of Threads At Runtime). Sometimes the number of iterations in a loop cannot be determined until runtime. If there is a negative performance impact for executing an OpenMP `parallel` region with multiple threads (e.g., a small number of iterations), specifying a minimum threshold will help maintain performance:

```
#pragma omp parallel for if (N >= threshold)
   for (i = 0; i < N; i++)
   {
      // Computation
   }
```

For this example code, the loop is only executed in parallel if the number of iterations exceeds the threshold specified by the programmer.

Fuse parallel loops with similar indices to improve granularity and data locality while minimizing overhead. In Figure 13 the first two loops (left-hand example code) can be easily merged (right-hand example code). Merging these loops increases the amount of work per iteration (i.e., granularity) and reduces loop overhead. The third loop is not easily merged because its iteration count is different. More important, however, a data dependence exists between the third loop and the previous two loops.

```
   for (j = 0; j < N; j++)
      a[j] = b[j] + c[j];

   for (j = 0; j < N; j++)
      d[j] = e[j] + f[j];

   for (j = 5; j < N – 5;
j++)
      g[j] = d[j+1] + a[j+1];
```

```
   for (j = 0; j < N; j++)
   {
      a[j] = b[j] + c[j];
      d[j] = e[j] + f[j];
   }

   for (j = 5; j < N – 5;
j++)
      g[j] = d[j+1] + a[j+1];
```

Figure 13. Fusing parallel loops with similar indices improves granularity and data locality

References

In this series, see also:

Intel® Software Development Tools, 2.4: Find Multithreading Errors With The Intel Thread Checker

Intel® Software Development Tools, 2.5: Using Thread Profiler To Evaluate OpenMP Performance

This chapter, 3.2: Granularity And Parallel Performance

This chapter, 3.3: Load Balance And Parallel Performance

This chapter, 3.6: Use Workload Heuristics To Determine Appropriate Number Of Threads At Runtime

4 Synchronization

In order to avoid race conditions during the execution of a threaded application, mutual exclusion to shared resources is required to allow a single thread to access and change the state of shared resources. The shared resource can be a data structure, or memory in the address space. Minimizing synchronization overheads is a critical to application performance. This chapter discusses effective synchronization practices in multithreaded applications.

In multithreaded applications, while a thread is executing a code section that accesses a shared resource (critical section), competing threads are either spinning or waiting in a queue. In order to ensure fairness in scheduling control over the lock among all competing threads, it is important to minimize the time spent by a thread within a critical section. This usually means reducing code within the critical section to the bare minimum to process the state change. The first topic in this chapter addresses design issues for optimally sized critical sections.

The standard threading implementations provide synchronization primitives that are optimized for the architecture, and have been widely tested in varying application scenarios. Typically, these primitives include optimized spin-waits, efficient scheduling algorithms, and as result, minimal synchronization, and scheduling overheads. Further, the synchronization primitives with the standard threading implementations are portable, usually are forward and backward compatible, and can be easily migrated across platforms. The second section in this chapter discusses the benefits of using standard threading APIs in preference to hand-coded synchronization functions.

The Windows* multithreading API provides multiple synchronization primitives – critical section, mutex, semaphore, events, and interlocked operations. All of these primitives implement mutual exclusion but have varying performance benefits and usage models. A comparison of the different synchronization primitives is discussed in the next chapter, “Memory Management.”

Most threading implementations provide non-blocking threading primitives as a cost-effective alternative to their blocking counterparts. The non-blocking threading calls are reviewed next.

The final section of this chapter deals with the merits of using double-check pattern locks to minimize lock-acquisition costs for events that are executed only once such as initialization, file opening/closing, dynamic memory allocation, etc.

4.1 Managing Lock Contention, Large And Small Critical Sections

Category

Synchronization

Scope

General multithreading

Keywords

Lock contention, synchronization, spin-wait, critical section, lock size

Abstract

In multithreaded applications, locks are used to synchronize entry to regions of code that access shared resources. The region of code protected by these locks is called a critical section. While one thread is inside a critical section, no other thread can enter. Therefore, critical sections serialize execution. This topic introduces the concept of critical section size – the length of time a thread spends inside a critical section – and its effect on performance.

Background

Critical sections ensure data integrity when multiple threads attempt to access shared resources. They also serialize the execution of code within the critical section. Threads should spend as little time inside a critical section as possible to reduce the amount of time other threads sit idle waiting to acquire the lock, or lock contention. In other words, it is best to keep critical sections small. On the other hand, using a multitude of small, separate critical sections can quickly introduce system overheads, from acquiring and releasing each separate lock, to such a degree that the performance advantage of multithreading is negated. In this latter case, one larger critical section could be best. Scenarios illustrating when it is best to use large or small critical sections will be explored below.

The thread function in Example Code 5 contains two critical sections. Assume that the critical sections protect different data and that the work in functions `DoFunc1` and `DoFunc2` is independent. We shall also assume that the amount of time to perform either of the update functions is always very small. The critical sections are separated by a call to `DoFunc1`. If the threads only spend a small amount of time in `DoFunc1`, the synchronization overhead of two critical sections may not be justified. In this case, a better scheme might be to merge the two small critical sections into one larger critical section, as in Example Code 6. If the time spent in `DoFunc1` is much higher than the combined time to execute both update routines, this might not be a viable option because the increased size of the critical section increases the likelihood of lock contention, especially as the number of threads increases.

```
Begin Thread Function ()
   Initialize ()

   BEGIN CRITICAL SECTION 1
      UpdateSharedData1 ()
   END CRITICAL SECTION 1

   DoFunc1 ()

   BEGIN CRITICAL SECTION 2
      UpdateSharedData2 ()
   END CRITICAL SECTION 2

   DoFunc2 ()
End Thread Function ()
```

Example Code 5. A threaded function containing two critical sections to protect updates to different shared data

```
Begin Thread Function ()
   Initialize ()

   BEGIN CRITICAL SECTION 1
      UpdateSharedData1 ()
      DoFunc1 ()
      UpdateSharedData2 ()
   END CRITICAL SECTION 1

     DoFunc2 ()
End Thread Function ()
```

Example Code 6. A threaded function containing one critical section that protects updates to all shared data used by the function

Let's consider a variation of the previous example. This time, assume the threads spend a large amount of time in the `UpdateSharedData2` function. Using a single critical section to synchronize access to `UpdateSharedData1` and `UpdateSharedData2`, as in Example Code 6, is no longer a good solution because the likelihood of lock contention is higher. On execution, the thread that gains access to the critical section spends a considerable amount of time in the critical section, while all the remaining threads are blocked. When the thread holding the lock releases it, one of the waiting threads is allowed to enter the critical section and all other waiting threads remain blocked for a long time. Therefore, two critical sections, as in Example Code 5, is a better solution for this case.

It is good programming practice to associate locks with particular shared data. Protecting all accesses of a shared variable with the same lock does not prevent other threads from accessing different shared variables protected by a different lock. Consider a shared data structure. A separate lock could be created for each element of the structure, or a single lock could be created to protect access to the whole structure. Depending on the

computational cost of updating the elements, either of these extremes could be a practical solution. The best lock granularity might lie somewhere in the middle. For example, given a shared array, a pair of locks could be used: one to protect the even numbered elements and the other to protect the odd numbered elements.

In the case where `UpdateSharedData2` requires a substantial amount of time to complete, dividing the work in this routine and creating new critical sections may be a better option. In Example Code 7, the original `UpdateSharedData2` has been broken up into two functions operating on different data. It is hoped that using separate critical sections will reduce lock contention. If the entire execution of `UpdateSharedData2` did not need protection, rather than enclose the function call, critical sections inserted into the function at points where shared data are accessed should be considered.

```
Begin Thread Function ()
   Initialize ()

   BEGIN CRITICAL SECTION 1
      UpdateSharedData1 ()
   END CRITICAL SECTION 1

   DoFunc1 ()

   BEGIN CRITICAL SECTION 2
      UpdateSharedData2 ()
   END CRITICAL SECTION 2

   BEGIN CRITICAL SECTION 3
      UpdateSharedData3 ()
   END CRITICAL SECTION 3

   DoFunc2 ()
End Thread Function ()
```

Example Code 7. Separating one critical section into two can help reduce lock contention

Advice

Balance the size of critical sections against the overhead of acquiring and releasing locks. Consider aggregating small critical sections to amortize locking overhead. Divide large critical sections with significant lock contention into smaller critical sections. Associate locks with particular shared data such that lock contention is minimized. The optimum probably lies somewhere between the extremes of a different lock for every shared datum and a single lock for all shared data.

Remember that synchronization serializes execution. Large critical sections indicate that the algorithm has little natural concurrency or that data partitioning among threads is sub-optimal. Nothing can be done about the former except changing the algorithm. For the latter, try to create local copies of shared data that the threads can access asynchronously.

The previous discussion of critical section size and lock granularity does not take the cost of context switching into account. When a thread blocks at a critical section waiting to acquire the lock, the operating system swaps an active thread for the idle thread. This is known as a context switch. In general, this is the desired behavior because it releases the CPU to do useful work. For a thread waiting to enter a small critical section, however, a spin-wait may be more efficient than a context switch. The waiting thread does not relinquish the CPU when spin-waiting. Therefore, a spin-wait is only recommended when the time spent in a critical section is less than the cost of a context switch.

Example Code 8 shows a useful heuristic to employ when using the Win32* threading API. This example uses the spin-wait option on Win32 `CRITICAL_SECTION` objects. A thread that is unable to enter a critical section will spin rather than relinquish the CPU. If the `CRITICAL_SECTION` becomes available during the spin-wait, a context switch is avoided. The spin-count parameter determines how many times the thread will spin before blocking. On uniprocessor systems the spin-count parameter is ignored. Code Sample 4 uses a spin-count of 1000 for each thread in the application but a maximum spin-count of 8000.

```
int gNumThreads;
CRITICAL_SECTION gCs;

int  main ()
{
   int spinCount = 0;
   ...
   spinCount = gNumThreads * 1000;
   if (spinCount > 8000) spinCount = 8000;
   InitializeCriticalSectionAndSpinCount (&gCs, spinCount);
   ...
}

DWORD WINAPI ThreadFunc (void *data)
{
   ...
   EnterCriticalSection (&gCs)
   ...
   LeaveCriticalSection (&gCs);
}
```

Example Code 8. Heuristic to control the behavior of waiting threads

Usage Guidelines

The spin-count parameter used in Example Code 8 should be tuned differently on Intel® processors with Hyper-Threading Technology. In general, spin-waits are detrimental to Hyper-Threading Technologyperformance. Unlike true symmetric multiprocessors (SMP), which have multiple physical CPUs, Hyper-Threading Technology creates two logical processors on the same CPU core. Spinning threads and threads doing useful work must compete for logical processors. Thus, spinning threads can impact the performance of multithreaded applications to a greater extent on Hyper-Threaded systems compared to SMP systems. The spin-count for the heuristic in Example Code 8 should be lower or not used at all.

References

In this series, see also:

Intel® Software Development Tools, 2.4: Find Multithreading Errors With The Intel Thread Checker

Intel® Software Development Tools, 2.5: Using Thread Profiler To Evaluate OpenMP Performance

Memory Management, 5.2: Use Thread-Local Storage To Reduce Synchronization

4.2 Use Synchronization Routines Provided By The Threading API Rather Than Hand-Coded Synchronization

Category

Synchronization

Scope

General multithreading

Keywords

Synchronization, spin-wait, Hyper-Threading, Win32 threads, OpenMP, Pthreads

Abstract

Application programmers sometimes write hand-coded synchronization routines rather than using constructs provided by a threading API in order to reduce synchronization overhead or provide different functionality than existing constructs offer. Unfortunately, using hand-coded synchronization routines may have a negative impact on performance, performance tuning, or debugging of multi-threaded applications.

Background

It is often tempting to write hand-coded synchronization to avoid the overhead sometimes associated with the synchronization routines from the threading API. Another reason programmers write their own synchronization routines is that those provided by the threading API do not exactly match the desired functionality. Unfortunately, there are serious disadvantages to hand-coded synchronization compared to using the threading API routines.

One disadvantage of writing hand-coded synchronization is that it is difficult to guarantee good performance across different hardware architectures and operating systems. The following example is a hand-coded spin lock written in C that will help illustrate these problems:

```
#include <ia64intrin.h>
void acquire_lock( int *lock )
{
   while
      (_InterlockedCompareExchange (lock, TRUE, FALSE) == TRUE );
}

void release_lock (int *lock)
{
   *lock = FALSE;
}
```

The `_InterlockedCompareExchange` compiler intrinsic is an interlocked memory operation which guarantees that no other thread can modify the specified memory location during its execution. It first compares the memory contents of the address in the first argument with the value in the third argument, and if a match occurs, stores the value in the second argument to the memory address specified in the first argument. The original value found in the memory contents of the specified address is returned by the intrinsic. In this example, the `acquire_lock` routine spins until the contents of the

memory location `lock` are in the unlocked state (`FALSE`) at which time the lock is acquired (by setting the contents of `lock` to `TRUE`) and the routine returns. The `release_lock` routine sets the contents of the memory location `lock` back to `FALSE` to release the lock.

Although this lock implementation may appear simple and reasonably efficient at first glance, it has several problems. First, if many threads are spinning on the same memory location, cache invalidations and memory traffic can become excessive at the point when the lock is released, resulting in poor scalability as the number of threads increases. Second, this code uses an atomic memory primitive which may not be available on all processor architectures, limiting portability. Third, the tight spin loop may result in poor performance for certain processor architecture features, such as Hyper-Threading Technology. Fourth, the `while` loop appears to the operating system to be doing useful computation, which could negatively impact the fairness of operating system scheduling. Although techniques exist for solving all these problems, they often complicate the code enormously, making it difficult to verify correctness. Tuning the code while maintaining portability is also difficult. These problems are better left to the authors of the threading API who have more time to spend verifying and tuning the synchronization constructs to be portable and scalable.

Another serious disadvantage of hand-coded synchronization is that it often decreases the accuracy of programming tools for threaded environments. For example, the Intel® Threading Tools need to be able to identify synchronization constructs in order to provide accurate information about performance (see Intel® Software Development Tools, 2.5: Using Thread Profiler To Evaluate OpenMP Performance) and correctness (see Intel Software Development Tools, 2.4: Find Multithreading Errors With The Intel Thread Checker) of the threaded application program. Threading tools are often designed to identify and characterize the functionality of the synchronization constructs provided by the supported threading API(s). Synchronization is difficult for the tools to identify and understand if standard synchronization API's are not used to implement it, which is the case in the example above. Sometimes tools support hints from the programmer in the form of tool-specific directives, pragmas, or API calls to identify and characterize hand-coded synchronization. Such hints, even if they are supported by a particular tool, may result in less accurate analysis of the application program than if threading API synchronization were used: the reasons for performance problems may be difficult to detect or threading correctness tools may report spurious race conditions or missing synchronization.

Advice

Avoid the use of hand-coded synchronization if possible. Instead, use the routines provided by your preferred threading API, such as `omp_set_lock/omp_unset_lock` or `critical/end critical` directives for OpenMP, `pthread_mutex_lock/pthread_mutex_unlock` for Pthreads, and `EnterCriticalSection/LeaveCriticalSection` or `WaitForSingleObject` or `WaitForMultipleObjects` and `ReleaseMutex` for the Win32 API. Study the threading API synchronization routines and constructs to find one that is appropriate for your application.

If a synchronization construct is not available that provides the needed functionality in the threading API, consider using a different algorithm for the program that requires less or different synchronization. Furthermore, expert programmers could build a custom synchronization construct from simpler synchronization API constructs instead of starting from scratch. If hand-coded synchronization must be used for performance reasons, consider using pre-processing directives to enable easy replacement of the hand-coded synchronization with a functionally equivalent synchronization from the threading API; thus increasing the accuracy of the threading tools.

Usage Guidelines

Programmers who build custom synchronization constructs from simpler synchronization API constructs should avoid using spin loops on shared locations to avoid non-scalable performance. If the code must also be portable, avoiding the use of atomic memory primitives is also advisable. The accuracy of threading performance and correctness tools may suffer because the tools may not be able to deduce the functionality of the custom synchronization construct, even though the simpler synchronization constructs from which it is built may be correctly identified.

References

In this series, see also:

Intel Software Development Tools, 2.4: Find Multithreading Errors With The Intel Thread Checker

Intel Software Development Tools, 2.5: Using Thread Profiler To Evaluate OpenMP Performance

This chapter, 4.3: Win32 Atomics Versus User-Space Locks Versus Kernel Objects For Synchronization

This chapter, 4.4: Use Non-Blocking Locks When Possible

See also:

John Mellor-Crummey, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors,” *ACM Transactions on Computer Systems,* 9, 21-65, 1991.

Intel Pentium® 4 and Intel® Xeon™ Processor Optimization Reference Manual, Chapter 7: “Multiprocessor and Hyper-Threading Technology,” Intel Developer Services.

4.3 Win32 Atomics Versus User-Space Locks Versus Kernel Objects For Synchronization

Category

Synchronization

Scope

Win32 multithreading

Keywords

Synchronization, lock contention, system overhead, mutual exclusion, Win32 threads

Abstract

When threads wait at a synchronization point, they are not doing useful work. Unfortunately, some degree of synchronization is usually necessary in multithreaded programs. The Win32 API provides several synchronization mechanisms with varying utility and system overhead.

Background

Synchronization constructs, by their very nature, serialize execution. However, very few multithreaded programs are entirely synchronization-free. Fortunately, it is possible to mitigate some of the system overhead associated with synchronization by choosing appropriate constructs. An increment statement (e.g., `var++`) will be used to illustrate the different constructs. If the variable being updated is shared among threads, the load→write→store instructions must be atomic (i.e., the sequence of instructions must not be preempted before completion). The Win32 API provides several mechanisms to guarantee atomicity, three of which are shown below:

```
#include <windows.h>

CRITICAL_SECTION cs;   /* Initialized  in main() */
HANDLE mtx;            /* CreateMutex called in main() */
static LONG counter= 0;

void IncrementCounter ()
{
   // Synchronize with Win32 interlocked function
   InterlockedIncrement (&counter);

   // Synchronize with Win32 critical section
   EnterCriticalSection (&cs);
      counter++;
   LeaveCriticalSection (&cs);

   // Synchronize with Win32 mutex
   WaitForSingleObject (mtx, INFINITE);
      counter++
   ReleaseMutex (mtx);
}
```

The advantages and disadvantages of each construct will now be discussed.

The Win32 interlocked functions (`InterlockedIncrement`, `InterlockedDecrement`, `InterlockedExchange`, `InterlockedExchangeAdd`, `InterlockedCompareExchange`) are limited to simple operations but they are faster than critical regions. In addition, fewer function calls are required. To enter and exit a Win32 critical region requires calls to `EnterCriticalSection` and `LeaveCriticalSection` or `WaitForSingleObject` and `ReleaseMutex`. The interlocked functions are also non-blocking whereas `EnterCriticalSection` and `WaitForSingleObject` (or `WaitForMultipleObjects`) block threads if the synchronization object is not available.

When a critical region is necessary, synchronizing on a Win32 `CRITICAL_SECTION` requires significantly less system overhead than Win32 mutex, semaphore, and event `HANDLE`s because the former is a user-space object whereas the latter are kernel-space objects. Though Win32 critical sections are usually faster than Win32 mutexes, they are not as versatile. Mutexes, like other kernel objects, can be used for inter-process synchronization. Timed-waits are also possible with the `WaitForSingleObject` and `WaitForMultipleObjects` functions. Rather than wait indefinitely to acquire a mutex the threads continue after the specified time limit expires. Setting the wait-time to zero allows threads to test whether a mutex is available without blocking. (Note that it is also possible to check the availability of a `CRITICAL_SECTION` without blocking using the `TryEnterCriticalSection` function.) Finally, if a thread terminates while holding a mutex, the operating system signals the handle to prevent waiting threads from becoming deadlocked. If a thread terminates while holding a `CRITICAL_SECTION`, threads waiting to enter this `CRITICAL_SECTION` are deadlocked.

A Win32 thread immediately relinquishes the CPU to the operating system when it tries to acquire a `CRITICAL_SECTION` or mutex `HANDLE` that is already held by another thread. In general, this is good behavior. The thread is blocked and the CPU is free to do useful work. Blocking and unblocking a thread is expensive, however. Sometimes it is better for the thread to try to acquire the lock again before blocking (e.g., on SMP systems, at small critical sections). Win32 `CRITICAL_SECTION`s have a user-configurable spin-count to control how long threads should wait before relinquishing the CPU. The `InitializeCriticalSectionAndSpinCount` and `SetCriticalSectionSpinCount` functions set the spin-count for threads trying to enter a particular `CRITICAL_SECTION`.

Advice

For simple operations on variables (i.e., increment, decrement, exchange) use fast, low-overhead Win32 interlocked functions.

Use Win32 mutex, semaphore, or event `HANDLE`s when inter-process synchronization or timed-waits are required. Otherwise, use Win32 `CRITICAL_SECTION`s, which have lower system overhead.

Control the spin-count of Win32 `CRITICAL_SECTION`s using the `InitializeCriticalSectionAndSpinCount` and `SetCriticalSectionSpinCount` functions. Controlling the how long a waiting thread spins before relinquishing the CPU is especially important for small and high-contention critical sections. Spin-count can significantly impact performance on SMP systems and CPUs with Hyper-Threading Technology.

Usage Guidelines

Beware of thread preemption when making successive calls to Win32 interlocked functions. For example, the two code segments in Figure 14 will not always yield the same value for `localVar` when executed with multiple threads. In the example using interlocked functions, thread preemption between any of the function calls can produce unexpected results. The critical section example is safe because both atomic operations (i.e., the update of global variable `N` and assignment to `localVar`) are protected.

```
static LONG N = 0;
LONG localVar;

InterlockedIncrement (&N);
InterlockedIncrement (&N);
InterlockedExchange
(&localVar, N);
```

```
static LONG N = 0;
LONG localVar;

EnterCriticalSection
(&lock);
  localVar = (N += 2);
LeaveCriticalSection
(&lock);
```

Figure 14. Fundamental differences between interlocked functions and critical sections

For safety, Win32 critical regions, whether built with `CRITICAL_SECTION` variables or mutex `HANDLE`s, should have only one point of entry and exit. Jumping into critical sections defeats synchronization. Jumping out of a critical section without calling `LeaveCriticalSection` or `ReleaseMutex` will deadlock waiting threads. Single entry and exit points also make for clearer code.

Prevent situations where threads terminate while holding `CRITICAL_SECTION` variables because this will deadlock waiting threads.

References

In this series, see also:

Intel Software Development Products, 2.3: Avoiding And Identifying False Sharing Among Threads With The VTune™ Performance Analyzer

This chapter, 4.2: Use Synchronization Routines Provided By The Threading API Rather Than Hand-Coded Synchronization

This chapter, 4.4: Use Non-Blocking Locks When Possible

See also:

Johnson M. Hart, *Win32 System Programming* (2nd Edition), Addison-Wesley, 2001

Jim Beveridge and Robert Wiener, *Multithreading Applications in Win32*, Addison-Wesley, 1997.

4.4 Use Non-Blocking Locks When Possible

Category

Synchronization

Scope

Windows threads, Pthreads, IA-32, Itanium® processor

Keywords

Non-blocking lock, synchronization, critical section, context switch, spin-wait

Abstract

Threads synchronize on shared resources by executing synchronization primitives offered by the supporting threading implementation. These primitives (such as mutex, semaphore, etc.) allow a single thread to own the lock, while the other threads either spin or block depending on their timeout mechanism. Blocking results in costly context-switch, whereas spinning results in wasteful use of CPU execution resources (unless used for very short duration). Non-blocking system calls, on the other hand, allow the competing thread to return on an unsuccessful attempt to the lock, and allow useful work to be done and thereby avoiding wasteful utilization of execution resources at the same time.

Background

Most threading implementations, including the Win32 and POSIX threads APIs provide both blocking and non-blocking thread synchronization primitives. Often the blocking primitives are used as default. When the lock attempt is successful, the thread gains control of the lock, and executes the code in the critical section. In the case of an unsuccessful attempt, however, a context-switch occurs and the thread is placed in a queue of waiting threads. A context-switch is costly and is avoidable for the following reasons:

- Context-switch overheads are considerable, especially if the threads implementation is based on kernel threads.
- Any useful work in the application following the synchronization call needs to wait execution until the thread gains control of the lock.

Using non-blocking system calls can alleviate the performance penalties. In this case, the application thread resumes execution following an unsuccessful attempt to lock the critical section. This avoids context-switch overheads and avoidable spinning on the lock. Instead, the thread performs useful work before a next attempt to gain control of the lock.

Advice

Use non-blocking synchronization functions to avoid context-switch overheads. Non-blocking synchronization functions usually start with 'try.' For instance, the Win32 API provides blocking and non-blocking critical sections:

```
void EnterCriticalSection (LPCRITICAL_SECTION cs);
bool TryEnterCriticalSection (LPCRITICAL_SECTION cs);
```

If the attempt to gain ownership of the critical section is successful, `TryEnterCriticalSection` returns the Boolean value `TRUE`. Otherwise, it returns `FALSE` and the thread can continue.

The following example shows a typical use of non-blocking synchronization:

```
CRITICAL_SECTION cs;

void threadfoo()
{
   while (TryEnterCriticalSection (&cs) == FALSE)
   {
      // Useful work
   }
   //
   // Code requiring protection by critical section
   //
   LeaveCriticalSection (&cs);
}
```

Similarly, Pthreads provides non-blocking versions of its mutex functions:

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

It is also possible to specify timeouts for Win32 synchronization primitives. The Win32 API provides the `WaitForSingleObject` and `WaitForMultipleObjects` functions to synchronize on kernel objects (i.e., `HANDLE`), e.g.:

```
DWORD WaitForSingleObject (HANDLE hHandle, DWORD dwMilliseconds);
```

where `hHandle` is the handle to the kernel object, and `dwMilliseconds` is the timeout interval after which the function returns if the kernel object is not signaled. A value of INFINITE indicates that the thread waits indefinitely. The thread waits until the relevant kernel object is signaled or a user-specified time interval has passed. Once the time interval elapses, the thread can resume execution. The following example demonstrates the use of `WaitForSingleObject` for non-blocking synchronization:

```
void threadfoo ()
{
   DWORD ret_value;
   HANDLE hHandle;

   ret_value = WaitForSingleObject (hHandle, 0);
   if (ret_value == WAIT_TIME_OUT)
   {
      // Thread could not acquire lock within the time interval
      //
      // Other useful work
      //
   }
   else if (ret_value == WAIT_OBJECT_0)
   {
      // Thread acquired lock within the time interval
      //
      // Code requiring protection by critical section
      //
   }
}
```

Similarly, `WaitForMultipleObjects` allows the thread to wait on the signal status of multiple kernel objects.

Usage Guidelines

When using non-blocking synchronization, for instance `TryEnterCriticalSection`, verify the return value to see if the request is successful before releasing the shared object.

References

In this series, see also:

Intel Software Development Products, 2.3: Avoiding And Identifying False Sharing Among Threads With The VTune™ Performance Analyzer

This chapter, 4.2: Use Synchronization Routines Provided By The Threading API Rather Than Hand-Coded Synchronization

This chapter, 4.3: Win32 Atomics Versus User-Space Locks Versus Kernel Objects For Synchronization

This chapter, 4.4: Use Non-Blocking Locks When Possible

This chapter, 4.5: Use A Double-Check Pattern To Avoid Lock Acquisition For One-Time Events

See also:

Aaron Cohen and Mike Woodring, *Win32 Multithreaded Programming*, O'Reilly and Associates, 1998.

Jim Beveridge and Robert Wiener, *Multithreading Applications in Win32 – the Complete Guide to Threads*, Addison Wesley, 1997.

Bil Lewis and Daniel J Berg, *Multithreaded Programming with Pthreads*, Sun Microsystems Press, 1998.

4.5 Use A Double-Check Pattern To Avoid Lock Acquisition For One-Time Events

Category

Synchronization

Scope

General multithreading

Keywords

Lock contention, synchronization, mutual exclusion, Win32 threads, Pthreads

Abstract

Acquiring locks, like synchronization, is an expensive operation. For one-time events (e.g., initialization, file opening/closing, dynamic memory allocation), it is often possible to use double-check locking (DCL) to avoid unnecessary lock acquisition.

Background

Synchronization, in this case lock acquisition, requires two interactions (i.e., locking and unlocking) with the operating system – an expensive overhead. When initializing a global, read-only table, for example, it is not necessary for every thread to perform the operation but every thread must check that the initialization occurred. For operations that are only executed once (e.g., initialization, file opening/closing, dynamic memory allocation), it is often possible to use DCL to avoid unnecessary lock acquisition. In DCL, if-tests are used to avoid locking after the first initialization, as the following pseudo-code illustrates:

```
Boolean initialized = FALSE

function InitOnce
{
   if not initialized
   {
      acquire lock
      if not initialized  ← Double-check!
      {
         perform initialization
         initialized = TRUE
      }
      release lock
   }
}
```

There are several interesting points about this pseudo-code. First, multiple threads can evaluate the first if-test as true. However, only the first thread to acquire the lock may perform the initialization and set the Boolean variable to true. When the lock is released, subsequent threads re-check the Boolean variable. Failure to double-check the Boolean control variable can result in re-initialization, possibly with different data, which could lead to unexpected results. Second, threads that call the function after initialization has occurred do not acquire the lock. The first if-test evaluates to false. Third, no thread can

return unless the initialization is complete. Finally, a data race exists for the Boolean variable. Specifically, a thread can read its value while another thread is modifying its value. This data race is benign because only the thread holding the lock can modify the variable. However, the Intel Thread Checker will still report storage conflicts on the Boolean variable (see 2.4: Find Multithreading Errors With The Intel Thread Checker).

Advice

Use DCL to avoid repeated lock acquisition when performing one-time operations. DCL is especially useful when threads repeatedly check whether the operation is complete. The following source code shows one way to implement DCL using C and the Win32 API:

```
#include <windows.h>

CRITICAL_SECTION lock;   /* Initialized elsewhere */
static int initialized = 0;

void init_once ()
{
   if (!initialized)
   {
      EnterCriticalSection (&lock);
      if (!initialized)
      {
         /* Perform initialization */
         initialized = 1;
      }
      LeaveCriticalSection (&lock);
   }
}
```

The following source code shows how to implement DCL using C and Pthreads:

```
#include <pthread.h>

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
static int initialized = 0;

void init_once ()
{
   if (!initialized)
   {
      pthread_mutex_lock (&lock);
      if (!initialized)
      {
         /* Perform initialization */
         initialized = 1;
      }
      pthread_mutex_unlock (&lock);
   }
}
```

For completeness, an example of DCL using OpenMP* is shown below:

```
subroutine init_once
   logical, save :: init = .FALSE.
   if (.not. init) then
      !$omp critical (once)
         if (.not. init) then
            ! Perform initialization
            init = .TRUE.
         endif
      !$omp end critical
   endif
end subroutine init_once
```

It is unlikely that DCL will ever be needed in an OpenMP program because OpenMP contains pragmas to express this capability (i.e., the `single` worksharing construct or the `master/barrier` combination).

Usage Guidelines

When initializing shared, read-only data, it is tempting to let multiple threads perform the initialization asynchronously. The initialization will be correct provided the threads are all writing the same values to the global data. However, asynchronous initialization could incur a serious performance penalty as multiple threads invalidate each other's cache lines.

The `pthread_once` function can be used in the same situations as DCL, but it has greater system overhead.

DCL should be used with caution in Java because some Java Virtual Machines implement the Java Memory Model incorrectly.

References

In this series, see also:

> Intel Software Development Products, 2.4: Find Multithreading Errors With The Intel Thread Checker
>
> This chapter, 4.3: Win32 Atomics Versus User-Space Locks Versus Kernel Objects For Synchronization
>
> This chapter, 4.4: Use Non-Blocking Locks When Possible

See also:

> Douglas C. Schmidt and Tim Harrison, "Double-Checked Locking", *Pattern Languages of Program Design 3* (Eds: Robert Martin, Frank Buschmann, and Dirke Riehle), Addison-Wesley, 1997.
>
> Brian Goetz, "Double-check locking: Clever, but broken" JavaWorld, February 2001.

5 Memory Management

Adding concurrency to applications can improve performance in obvious ways. Other chapters in this series address many of the issues that can impact the performance of threaded applications. Avoiding contention for heap resources, using storage that is local to threads rather than shared to reduce synchronization, and carefully managing memory allocations are some of the less obvious, but no less important, considerations that can also impact threaded performance. These memory management issues are covered in this chapter.

5.1 Avoiding Heap Contention Among Threads

Category

Memory Management

Scope

General multithreading

Keywords

Heap contention, synchronization, dynamic memory allocation, lock contention, stack allocation

Abstract

Allocating memory from the system heap can be an expensive operation. To make allocation thread-safe, a lock is used to synchronize access to the heap. The contention on this lock can limit the performance benefits from multithreading. To solve this problem, change the allocation strategy to avoid using shared lock.

Background

The system heap (as used by `malloc`) is a shared resource. To make it safe to use by multiple threads it is necessary to add synchronization to gate access to the shared heap. Synchronization, in this case lock acquisition, requires two interactions (i.e., locking and unlocking, with the operating system – an expensive overhead.

The OpenMP* implementation in the Intel® 7.0 compilers exports two functions, `kmp_malloc` and `kmp_free`. These functions maintain a per-thread heap attached to each thread of the OpenMP team. Threads that call these functions avoid the use of the lock that protects access to the standard system heap. The `threadprivate` directive can be used as well to create a private copy of globally declared variables for each thread in the OpenMP team.

The Win32* `HeapCreate` function can be used to allocate separate heaps for all of the threads used by the application. The flag `HEAP_NO_SERIALIZE` is used to disable the use of synchronization on this new heap since only a single thread will access it.

If the heap handle is stored in a Thread Local Storage (TLS) location, this heap can be used whenever an application thread needs to allocate or free memory. Note that memory allocated in this manner must be explicitly released by the same thread that performs the allocation. For Pthreads applications, the `pthread_key_create` and `pthread_{get|set}specific` API can be used to obtain access to TLS but the management of this global storage is the programmer's responsibility.

If you need to use a more general replacement (where the thread which allocates the memory is not necessarily the thread which releases the memory, then it may be more appropriate to look into using a commercial replacement to the heap manager as listed in the references section.

The following example uses several features of the Win32 API:

```
#include <windows.h>

static DWORD tls_key;

__declspec (dllexport) void* thr_malloc (size_t n)
{
   return HeapAlloc (TlsGetValue (tls_key), 0, n);
}

__declspec (dllexport) void thr_free (void *ptr)
{
   HeapFree (TlsGetValue (tls_key), 0, ptr);
}

BOOL WINAPI DllMain (HINSTANCE hinstDLL,
                     DWORD fdwReason,
                     LPVOID lpReserved)
{
   switch (fdwReason)
   {
      case DLL_PROCESS_ATTACH:
         // Use Thread Local Storage to remember the heap
         tls_key = TlsAlloc ();
         TlsSetValue (tls_key, GetProcessHeap ());
         break;

      case DLL_THREAD_ATTACH:
         // Use HEAP_NO_SERIALIZE to avoid lock contention
         TlsSetValue
               (tls_key, HeapCreate (HEAP_NO_SERIALIZE, 0, 0));
         break;

      case DLL_THREAD_DETACH:
         HeapDestroy (TlsGetValue (tls_key));
         break;

      case DLL_PROCESS_DETACH:
         TlsFree (tls_key);
         break;
    }
    return TRUE;   // Successful DLL_PROCESS_ATTACH
}
```

First, it uses a dynamic load library (DLL) to allow the threads to be registered at the point of creation. It also uses TLS to remember the heap that is assigned to each thread. Finally, it uses the ability of the Win32 API to independently manage unsynchronized heaps.

Advice

In addition to the use of multiple independent heaps, it is also possible to incorporate other techniques to minimize the lock contention caused by a shared lock that is used to protect the system heap. If the memory is only to be accessed within a small lexical context, the `alloca` routine can sometimes be used to allocate memory from the current stack frame. This memory is automatically deallocated upon function return.

A per-thread free list is another technique. Initially, memory is allocated from the system heap with `malloc`. When the memory would normally be released it is added to a per-thread linked-list. If the thread needs to reallocate memory of the same size, it can immediately retrieve the stored allocation from the list without going back to the system heap.

```
struct MyObject
{
   struct MyObject *next;
};

static __declspec(thread) struct MyObject *freelist_MyObject = 0;

struct MyObject *malloc_MyObject ()
{
   struct MyObject *p = freelist_MyObject;

   if (p == 0)
      return malloc (sizeof (struct MyObject));

   freelist_MyObject = p->next;
   return p;
}

void free_MyObject (struct MyObject *p)
{
   p->next = freelist_MyObject;
   freelist_MyObject = p;
}
```

Usage Guidelines

With any optimization you encounter trade-offs. In this case the trade-off is in exchanging lower contention on the system heap for higher memory usage. When each thread is maintaining its own private heap or collection of objects, these areas are not available to other threads. This may result in a “memory imbalance” between the threads, similar to the “load imbalance” you encounter when threads are performing varying amount of work (see Application Threading, 3.3: Load Balance And Parallel Performance). The memory imbalance may cause the working set size to increase and the total memory usage by the application to also increase. The increase in memory usage usually has a minimal performance impact. An exception occurs when the increase in memory usage exhausts the available memory. If this happens it may cause the application to either abort or swap to disk.

References

In this series, see also:

Intel Software Development Products, 2.3: Avoiding And Identifying False Sharing Among Threads With The VTune™ Performance Analyzer

Intel Software Development Products, 2.4: Find Multithreading Errors With The Intel Thread Checker

Synchronization, 4.1: Managing Lock Contention, Large And Small Critical Sections

See also:

MicroQuill SmartHeap for SMP

The HOARD memory allocator

Documentation for the following Win32 functions:

HeapAlloc, HeapCreate, HeapFree

TlsAlloc, TlsGetValue, TlsSetValue

Alloca

5.2 Use Thread-Local Storage To Reduce Synchronization

Category

Memory Management

Scope

General multithreading

Keywords

Thread-local storage, synchronization, OpenMP, Pthreads, Win32 threads

Abstract

Synchronization is often an expensive operation that can limit the performance of a multi-threaded program. Using thread-local data structures instead of data structures shared by the threads can reduce synchronization in certain cases, thus allowing a program to run faster.

Background

When data structures are shared by a group of threads and at least one thread is writing into them, synchronization between the threads is sometimes necessary to make sure that all threads see a consistent view of the shared data at all times. The typical synchronized access regime for threads in this situation is for a thread to acquire a lock, read or write the shared data structures, then release the lock.

All forms of locking have overhead to maintain the lock data structures and they use atomic instructions that slow down modern processors. Synchronization also slows down the program because it eliminates parallel execution inside the synchronized code, forming a serial execution bottleneck. Therefore, when synchronization occurs within a time-critical section of code, code performance can suffer.

The synchronization can be eliminated from the multithreaded, time-critical code sections if the program can be re-written to use thread-local storage instead of shared data structures. This is possible if the nature of the code is such that real-time ordering of the accesses to the shared data is unimportant. Synchronization can also be eliminated when the ordering of accesses is important, if the ordering can be safely postponed to execute during infrequent, non-time-critical sections of code.

Consider, for example, the use of a variable to count events that happen on several threads. The following code shows one way to write such a program in OpenMP:

```
int count=0;

#pragma omp parallel shared(count)
{
   if (event_happened)
   {
      #pragma omp atomic
         count++;
   }
}
```

This program pays a price each time the event occurs because it must synchronize to guarantee that only one thread at a time increments `count`. Every event causes synchronization. Removing the synchronization makes the program run faster. One way to do this safely is to have each thread count its own events in the parallel region then sum the individual counts later. The following code demonstrates this technique:

```
int count=0;
int tcount=0;
#pragma omp threadprivate(tcount)

#pragma omp parallel
{
   if (event_happened)
   {
      tcount++;
   }
}

#pragma omp parallel shared(count)
{
   #pragma omp atomic
      count += tcount;
}
```

This program uses a `tcount` variable that is private to each thread to store the count for each thread. After the first parallel region counts all the local events, a subsequent region adds this count into the overall count. This solution trades synchronization per event for synchronization per thread. Performance will improve if the number of events is much larger than the number of threads.

An additional advantage of using thread-local storage during time-critical portions of the program is that the data may stay live in a processor's cache longer than shared data, if the processors do not share a data cache. When the same address exists in the data cache of several processors and is written by one of them, it must be invalidated in the caches of all other processors, causing it to be re-fetched from memory when the other processors access it. But thread-local data will never be written by any other processors and will therefore be more likely to remain in the cache of its processor.

The previous example code shows one way to specify thread-local storage in OpenMP. To do the same thing with Pthreads, the programmer must create a key to access thread-local storage, e.g.:

```
#include <pthread.h>

pthread_key_t tsd_key;
<arbitrary data type> value;

if (pthread_key_create (&tsd_key, NULL))
   err_abort(status, “Error creating key”);

if (pthread_setspecific( tsd_key, value))
   err_abort(status, “Error in pthread_setspecific”);

value = (<arbitrary data type>)pthread_getspecific( tsd_key );
```

With the Win32 API, the programmer allocates a TLS index with `TlsAlloc` then uses that index to set a thread-local value, e.g.:

```
DWORD tls_index;
LPVOID value;

tls_index = TlsAlloc();

if (tls_index == TLS_OUT_OF_INDEXES)
   err_abort( tls_index, “Error in TlsAlloc”);

status = TlsSetValue( tls_index, value );

if (status == 0)
   err_abort( status, “Error in TlsSetValue”);

value = TlsGetValue (tls_index);
```

In OpenMP, one can also create thread-local variables by specifying them in a `private` clause on the `parallel` pragma or the `threadprivate` pragma. These variables are automatically deallocated at the end of the parallel region. Of course, another way to specify thread-local data, regardless of the threading model, is to use variables allocated on the stack in a given scope. Such variables are deallocated at the end of the scope.

Advice

The technique of thread-local storage is applicable if synchronization is coded within a time-critical section of code, and if the operations being synchronized need not be ordered in real-time. If the real-time order of the operations is important, then the technique can still be applied if enough information can be captured during the time-critical section to reproduce the ordering later, during a non-time-critical section of code.

Consider the following example where threads write data into a shared buffer:

```
int buffer[ENTRIES];

main()
{
   #pragma omp parallel
   {
      update_log (time, value1, value2);
   }
}

void update_log (time, value1, value2)
{
   #pragma omp critical
   {
      if (current_ptr + 3 > ENTRIES)
      {
         print_buffer_overflow_message ();
      }
      buffer[current_ptr] = time;
      buffer[current_ptr+1] = value1;
      buffer[current_ptr+2] = value2;
      current_ptr += 3;
   }
}
```

Let's assume that `time` is some monotonically increasing value and the only real requirement of the program for this buffer data is that it be written to a file occasionally sorted according to `time`. We can eliminate the synchronization in the `update_log` routine by using thread-local buffers. Each thread allocates a separate copy of `tpbuffer` and `tpcurrent_ptr`. This allows us to eliminate the critical section in `update_log`. The entries from the various thread-private buffers can be merged later, in a non-time-critical portion of the program.

Usage Guidelines

One must be careful about the trade-offs involved in this technique. The technique does not remove the need for synchronization. It only moves the synchronization from a time-critical section of the code to a non-time-critical section of the code. First, determine whether the original section of code containing the synchronization is actually being slowed down significantly by the synchronization. (The Intel® VTune™ Performance Analyzer can be used to generate a performance profile.) Second, determine whether the time ordering of the operations is critical to the application. If not, synchronization can be removed, as in the event-counting code. If time ordering is critical, can the ordering be correctly re-constructed later? Third, verify that moving synchronization to another place in the code will not cause similar performance problems in the new location. One way to do this is to show that the number of synchronizations will decrease dramatically because of your work (such as in the event-counting example above).

References

In this series, see also:

Intel Software Development Products, 2.4: Find Multithreading Errors With The Intel Thread Checker

Intel Software Development Products, 2.5: Using Thread Profiler To Evaluate OpenMP Performance

Application Threading, 3.5: Expose Parallelism By Avoiding Or Removing Artificial Dependencies

See also:

David R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.

Johnson M. Hart, *Win32 System Programming* (2nd Edition), Addison-Wesley, 2001.

Jim Beveridge and Robert Weiner, *Multithreading Applications in Win32*, Addison-Wesley, 1997.

5.3 Offset Thread Stacks To Avoid Cache Conflicts On Intel® Processors With Hyper-Threading Technology

Category

Memory Management

Scope

Multithreading with Pthreads or the Win32 API on Intel® processors with Hyper-Threading Technology

Keywords

Hyper-Threading Technology, cache-coherence, data alignment, VTune, stack allocation

Abstract

Hyper-Threading Technology-enabled processors share the first-level data cache on a cache-line basis among the logical processors. Frequent accesses to the virtual addresses on cache lines modulo 64 KB apart can cause alias conflicts that negatively impact performance. Since thread stacks are generally created on modulo 64 KB boundaries, accesses to the stack often conflict. By adjusting the start of the stack, the conflicts can be reduced and result in significant performance gains. Note that the 64 KB alias conflict is processor implementation dependent. Future processors may adjust the modulo boundary or eliminate this conflict altogether.

Background

Intel processors with Hyper-Threading Technology share the first-level data cache among logical processors. Cache lines whose virtual addresses are modulo 64 KB apart will conflict for the same slot in the first-level data cache. This can both affect the first- level data-cache performance and impact the branch prediction unit. In addition to 64 KB alias conflicts, it is possible to increase the number of branch mispredictions when the processor core logic uses speculative data with addresses modulo one megabyte apart. Under Microsoft Windows operating systems, thread stacks are currently created on a multiple of one-megabyte boundaries by default. Two threads with very similar stack frame images and access patterns to local variables on the stack are very likely to cause alias conflicts resulting in substantial degradation. Future implementations of Intel processors with Hyper-Threading Technology will likely address both sources of alias conflicts. Adjusting the initial thread stack address of each thread is a simple workaround that can restore considerable performance to your application on Intel processors with Hyper-Threading Technology.

Advice

Create a stack offset for each thread to avoid first-level data cache-line conflicts between threads on Hyper-Threading-enabled processors.

There are two ways to determine if your application performance on Hyper-Threading enabled processors is suffering from these alias conflicts. The first, and most definitive, method is to try the suggested work-around across your application's performance workloads. By comparing the resulting performance with and without Hyper-Threading technology enabled, you can directly measure the relative performance difference. The second method is to use the Intel VTune Performance Analyzer. You will need to collect

both clock tick events as well as 64 KB alias-conflict events across your application's performance workloads with and without Hyper-Threading Technology enabled. After sorting the modules and functions in your application by clock ticks from highest to lowest, compare the number of 64 KB alias events. It's not unusual to see an increase on the order of three times the number of 64 KB alias events with Hyper-Threading technology enabled. However, applications with a difference of eight times or greater at a module or function level have been shown to improve performance significantly using the optimization described below. If a sizeable portion of the total execution time is spent in the module or function, this will translate directly to an overall application level performance improvement.

Note that enabling or disabling Hyper-Threading support in Intel processors requires support in the system BIOS. Some BIOS implementations between vendors may not support user level access to enable or disable the Hyper-Threading feature.

Typically, threads are created using an operating system-specific application interface and passing it a pointer to a function as well as a pointer to a block of data specific to the thread. The key to adjusting the initial thread stack address is to replace the original function pointer with an intermediate function that can adjust the stack by a variable amount depending on the number of threads created. A new intermediate parameter block is needed that contains a pointer to the original thread function, a thread id, and a pointer to the original parameter data block. The intermediate function can adjust the stack address and then call the original function passing on the original thread specific parameter data. Using the new parameter block with a function pointer is a generic implementation that can be used for a pool of threads that may need to invoke different functions for a thread. As a less general alternative, you could avoid the function pointer technique and have the intermediate function call the original function directly. However, be careful that the compiler does not in-line the original thread function within the alternative thread function. If the original thread function is 'in-lined', the benefit of the adjusted stack address for the original function is lost. Using the intermediate function method with a function pointer avoids this possibility because the compiler cannot determine which function to in-line at compile time.

The easiest way to adjust the initial stack address for each thread is to call the memory allocation function, `_alloca`, with varying byte amounts in the intermediate thread function. The `_alloca` function allocates memory directly on the stack. By adjusting the number of bytes passed to the `_alloca` function, you can adjust the next function's starting stack address. The `_alloca` function is found in the `malloc.h` header file. Using this technique to adjust the stack address is allocating virtual memory in each thread's stack frame that will go unused. In Example Code 9, a one kilobyte offset multiplied by the thread ID number is used to offset the thread stack frames. One kilobyte is not a magic number but one that has generally worked across various applications. One important point to note is that current versions of Microsoft Windows* operating systems have a limit on the amount of virtual memory accessible to a given process. If the limit on virtual memory is an important consideration for your application, you will need to determine the best offset or modify this technique within this constraint.

```
// Original thread parameter data structure
struct ParameterBlk
{
   int thread_specific_data;

// Padding to keep thread data at least a cache-line apart
   char padding[2 * CACHE_LINE_SZ – sizeof (int)];
};

typedef DWORD (*PFI) (void*);

// Structure containing arguments provided to each thread
struct FunctionBlk
{
   PFI ThreadFuncPtr;
   struct ParameterBlk* function_parameters;
   unsigned int thread_number;

   // Padding to keep thread data at least a cache-line apart
   char padding[2 * CACHE_LINE_SZ – sizeof (PFI) -
                                sizeof(struct ParameterBlk*) -
                                sizeof(unsigned int)];
};

DWORD WINAPI OriginalThreadProc (LPVOID ptr)
{
   // This would have been the original thread function
   return 0;
}

#define STACK_OFFSET 1024

DWORD WINAPI IntermediateThreadProc (LPVOID ptr)
{
   struct FunctionBlk* parameter = (struct FunctionBlk*) ptr;
   // Adjusting stack address
   _alloca (parameter->thread_number * STACK_OFFSET);

   // Calling original thread procedure using a function pointer.

   // You could call the function directly as shown blow but be
   // careful that the function doesn't get inlined.
   return
     (*parameter->ThreadFuncPtr)(parameter->function_parameters);
}
```

Example Code 9. Offsetting thread stacks with `_alloca` can avoid cache conflicts

When determining how many threads to create, you should consider using the main thread to do a portion of the work. The main thread is already likely to have a very different stack frame image and data-access pattern from the child threads that start with a clean stack frame aligned on one megabyte boundaries. Plus, there is one less child

thread to synchronize and manage. Note that this may not be desirable if the main thread must manage other tasks or be responsive to user input.

Usage Guidelines

A single-source implementation of the thread stack offsets can be used for multi-processor systems without performance impact. However, use of the stack offset can reduce the overall virtual memory available to an application. In general, this will affect only very large applications with a large number of threads. By adjusting the stack offset amount, you can balance performance needs versus virtual memory.

The best size for the stack offset is application-dependent. Thread functions that have deep thread stacks due to local variables with subsequent function calls or that operate on large local data structures within a loop tend to perform better with a larger stack-offset size. Conversely, thread functions with smaller stack sizes can perform well with a smaller stack offset. In general, increments of one kilobyte stack offsets per thread have worked well for many applications.

References

In this series, see also:

> Intel Software Development Products, 2.3: Avoiding And Identifying False Sharing Among Threads With The VTune™ Performance Analyzer
>
> This chapter, 5.1: Avoiding Heap Contention Among Threads

See also:

> "Adjusting Thread Stack Address To Improve Performance on Intel® Xeon™ Processors," Phil Kerly (http://developer.intel.com)